Implémentation en C de l'arithmétique de Sisyphe

José Grimm

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Implémentation en C de l'arithmétique de Sisyphe

José Grimm

N° 168

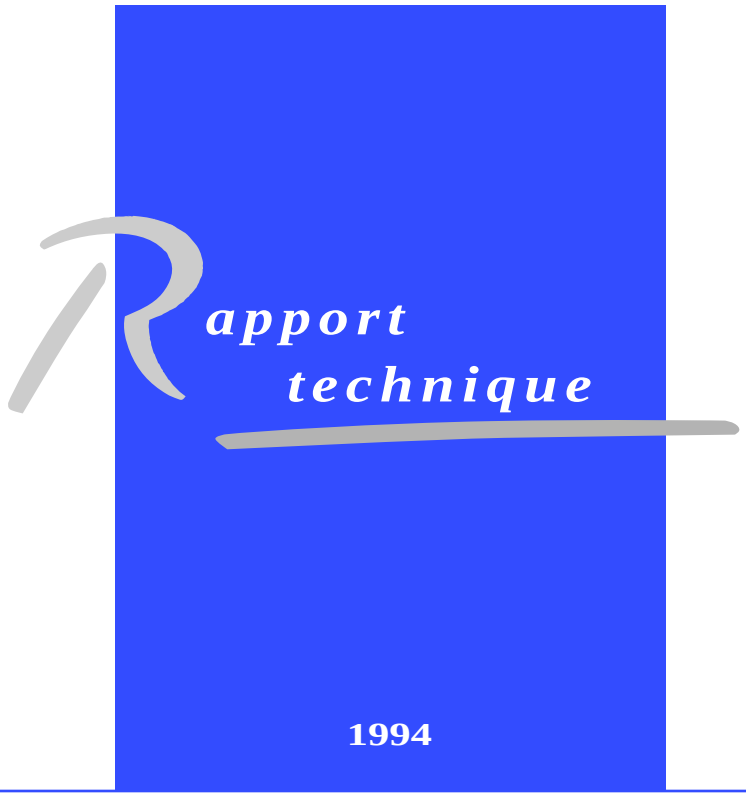
Novembre 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel



ISSN 0249-6399

Implémentation en C de l'arithmétique de Sisyphe

José Grimm

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet SAFIR *

Rapport technique n ° 168 — Novembre 1994 — 146 pages

Résumé : Ce rapport décrit l'implémentation de l'arithmétique dans le système de calcul formel SISYPHE. Toutes les procédures sont écrites dans le langage C, et supposent l'existence d'un gestionnaire de mémoire conservateur. La plupart des algorithmes décrits ici ont été implémentés initialement en Lisp ou dans le langage de SISYPHE.

Dans une première partie nous définissions les primitives de bas niveau pour implémenter une arithmétique sur les entiers positifs de taille arbitraire. Dans une seconde partie, nous définissons les entiers signés et les fractions rationnelles. Finalement nous introduirons les nombres complexes et les fonctions transcendantes et décrirons les fonctions d'entrées-sorties. Comme application, nous donnons un algorithme de factorisation des entiers (méthodes de Pollard, Morrison et Brillhart, courbes elliptiques).

Mots-clé : arithmétique, C, gestionnaire de mémoire, factorisation, pgcd

(Abstract: pto)

∗. SAFIR est un projet commun au CNRS, à l'INRIA et à l'UNSA

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Implementation in C of the arithmetic of Sisyphe

Abstract: This report describes the implementation of the arithmetics of the computer algebra system SISYPHE, written in C, which relies on the existence of a conservative garbage collector. Most algorithms were already written in Lisp, or the language of SISYPHE and have been translated.

In a first part, we define and use low level primitives to implement unsigned bignums of arbitrary size. In a second part, we extend these to signed integers and rational numbers. Finally, in the last part complex numbers and transcendental functions are defined and I/O functions are described. As an application, we give an algorithm to factor integers, using methods of Pollard, Morrison and Brillhart or elliptic curves.

Key-words: arithmetic, C, memory management, factorisation, gcd

Chapitre 1

Introduction

Ce rapport décrit l'implémentation de l'arithmétique rationnelle de SISYPHE [1] faite en C. Dans une première partie, nous décrirons les algorithmes manipulant les nombres entiers en précision arbitraire. Dans les chapitres suivants, nous décrirons l'arithmétique générique et certains algorithmes particuliers.

1.1 Historique

Ce logiciel a été écrit en 1987 pour servir de base pour un système de calcul formel, qui a reçu plus tard (en 1988) le nom de SISYPHE. Le système SISYPHE est actuellement développé par le projet SAFIR, projet commun à l'INRIA, au CNRS et à l'Université de Nice-Sophia Antipolis.

La première implémentation de ce logiciel a été faite dans la version 15 de LELISP, elle était entièrement écrite en LLM3, et utilisait des optimisations spécifiques sur Multics. A l'époque, il y avait dans LELISP une seule implémentation de l'arithmétique générique (qui existe encore, et décrite dans le manuel de référence de LELISP [7, chapitre 10]). Cette représentation utilise des arbres équilibrés pour représenter des entiers (il s'agit de cellules de listes étiquettées ou *tcons*), et a comme principal avantage d'être extrêmement efficace sur les entiers de taille intermédiaire (quelques mots machines), et assez efficace pour les fonctions comme l'addition et la comparaison. Par contre, elle est assez lente pour les calculs de pgcd. Or, il est arrivé, lors de la factorisation de polynômes, de calculer un pgcd sur des entiers ayant plusieurs milliers de chiffres. Dans ce cas cette implémentation donne des résultats catastrophiques: plusieurs *milliers* de fois plus lent que notre arithmétique [voir la table section 2.6].

Il devint évident en 1992 qu'il fallait adopter une représentation des entiers sous forme de suite de bits (vecteur de chiffres ou chaînes de caractères). La question qui se posait était de savoir s'il fallait ou non étendre les primitives Lisp. Comme cela n'a pas été fait, il restait deux solutions.

La première solution consistait à prendre les primitives actuelle de Lisp. C'est l'option choisie en 1992, le code LLM3 a alors été traduit en Lisp, sans perte d'efficacité notable (ceci est essentiellement dû au rajout dans LELISP d'un nouveau compilateur, COMPLICE, nettement plus efficace que le compilateur par défaut), et une première version de ce document [2] a été écrite pour servir de base à une extension future.

La seconde stratégie consistait à écrire de nouvelles primitives, plus puissantes (par exemple, une fonction qui additionne deux vecteurs de chiffres) en assembleur, ou plus généralement dans un langage comme C. Cette stratégie a été utilisée dans [3]. L'ensemble de ces primitives recouvre à peu près le chapitre 2, et doit être complété par des fonctions Lisp, telles que celles décrites dans les chapitres suivants.

1.2 L'implémentation en C

Nous avons décidé en 1994 une traduction complète de cette arithmétique dans le langage C. Les algorithmes restent inchangés par rapport à la version précédente, avec deux ajouts majeurs: utilisation de la méthode de Karatsuba pour la multiplication, et ajout d'une fonction qui calcule a^b modulo c de façon efficace. Un certain nombre de fonctionalités utilisées par ailleurs dans SISYPHE telles que fractions continues, racines carrées, etc., ont été rajoutées dans les chapitres 3 et 4. Un chapitre 5 décrivant la factorisation des entiers a été ajouté.

L'un des avantages par rapport à une implémentation en LLM3 est que l'arithmétique de base est sur 32bits et non sur 16bits, le second est que ce logiciel devient indépendant de LLM3, est n'est donc pas lié à l'avenir de la version 15 de LELISP. On supprime également une limitation importante sur la taille des objets: dans LELISP, les indices sont limités à 15bits, ce qui donne un maximum de 512 000 bits pour un nombre dans l'implémentation précédente de SISYPHE, et la moitié de ce chiffre pour l'implémentation de [3], qui utilise des chaînes de caractères pour y ranger des entiers 32bits.

Contrairement à [3], ce logiciel n'est pas entièrement portable: il suppose d'une part l'existence d'un gestionnaire de mémoire, et d'autre part un mécanisme d'entrées/sorties hérité de LELISP. Ce mécanisme est utilisé pour lire et écrire des nombres. On peut facilement supprimer la fonction qui lit des nombres, et n'utiliser que la fonction qui convertit une chaîne de caractères en nombre. Par ailleurs on utilise une seule fonction d'impression, la fonction `princn` qui envoie un octet sur le flux de sortie courant. On peut la remplacer par une fonction qui écrit dans une chaîne donnée (et qui s'occupe de la gestion de la mémoire utilisée pour ceci).

L'existence d'un gestionnaire de mémoire intervient de deux façons différentes: dans le chapitre 2, nous supposons que les fonctions utilisables par ailleurs travaillent en place sur une copie des arguments; il suffit donc d'avoir une fonction qui copie une suite de chiffres. Dans les chapitres suivants, on introduit des objets typés (entiers, rationnels, etc.), il faut pouvoir les créer, les détruire et tester leur types. On introduit également des fonctions utilisateur, à nombre fixe ou variable d'arguments, et des variables-fonctions. Ce dernier point est très fortement lié à l'implémentation de l'évaluateur Lisp, nous ne le décrirons pas ici.

1.3 Les structures de données

Note code repose essentiellement sur l'existence d'un gestionnaire de mémoire conservateur. En d'autres termes, on alloue explicitement de la mémoire, et le gestionnaire la libère si elle n'est plus utilisée. Le qualificatif 'conservateur' appliqué au gestionnaire signifie qu'il n'est pas besoin de lui dire si une variable x contient une adresse ou un chiffre (en interne, ce sont deux objets représentés sur 32bits, et il peut y avoir conflit). Pour minimiser ces conflits, deux structures de données sont utilisées: vecteurs de pointeurs et vecteurs de chiffres. Dans le second cas, on prévient

le gestionnaire que l'objet ne contient pas de pointeurs. Il faut donc faire une double indirection pour accéder à un chiffre. Dans le cas d'un vecteur de pointeurs, une seule indirection suffit (dans le cas de LeLispv15, une double indirection est toujours nécessaire).

Deux types d'entiers sont manipulés par le programme. Il s'agit d'une part de chiffres (*unsigned int* en C), des entiers non signés 32bits, et des vecteurs de chiffres. Ces objets ne sont pas visibles par l'utilisateur. D'autre part l'utilisateur fournit et reçoit des petits ou grands entiers. Dans le second cas, il s'agit d'un nombre 32bits signé, qui est un objet typé. En particulier, créer un tel entier peut allouer de la mémoire, et comparer deux entiers via leur adresse ne suffit pas. Par contre, une table des petits entiers est créée lors de l'initialisation du programme, ce qui assure en particulier l'unicité de la représentation de 0 et 1.

L'accès aux vecteurs de pointeurs se fait à l'aide des macros `VREF` et `VSET`, ce qui rend la structure des objets transparente au programmeur. La double indirection pour accéder ou modifier un chiffre se fait à l'aide des macros `get_digit`, `set_digit`, et le pointeur de tas s'obtient par la macro `UHEAP`. La fonction `make_int` crée un petit entier, et la macro `Int_val` rend l'entier C associé à un petit entier Lisp. Il y a aussi `make_double` et `Double_val` pour les flottants.

Pour simplifier le code, l'arithmétique ne manipule pas toutes les structures de données numériques de SISYPHE ou HYPERION (BigFloat, polynômes ou matrices de polynômes à coefficients réels ou complexes). Les quatre opérations (addition, soustraction, multiplication, division, ou division entière) ne posent pas trop de problème: en cas d'erreur de type, on appelle une fonction `arith_err`, qui pour l'instant se contente de signaller une erreur, mais qui peut très bien calculer un type commun, convertir les arguments dans ce type, et faire l'opération dans ce type. Les autres fonctions sont plus compliquées. Par exemple, la fonction arc sinus, appliquée à l'entier 2 rend un nombre complexe. On pourrait trés bien rendre également un BigFloat avec une certaine précision. De même, si P est un polynôme, la fonction sinus appliquée à P pourrait rendre les premiers termes du développement en série de Taylor (qui est un polynôme).

1.4 Algorithmes

L'ensemble du code utilisé pour implémenter l'arithmétiques est regroupé en un certain nombre d'algorithmes. Chaque algorithme est formé de procédures (fonctions C, qui peuvent rendre ou non une valeur), de macros ou de fonctions. Une macro est comme une procédure, mais en général le code est très court et ce code (après substitution des paramètres) remplace l'appel de la macro. Par exemple, la macro `HIBITS`(x) décale le chiffre x de 16bits vers la droite et rend x. Le code C de la macro est `((x)>>16)`. Quand on dit: « décrémenter x de `HIBITS`(y) », le code C généré est `x -= (y)>>16`. Un autre avantage des macros est le suivant: si `HILO`(x, a, b) est défini comme: « poser $a =$ `HIBITS`(x), $b =$ `LOBITS`(x) », un appel du genre `HILO(X,A,B)` change a et b, les variables locales de la procédure. Par contre, s'il s'agit d'une macro, on modifie A et B. La première expansion donne se transforme en `A=HIBITS(X)`, `B=LOBITS(X)`. La seconde donne `A=X>>16`, `B=X&x7FFF` (et dans une troisième phase `x7FFF` est remplacé par sa valeur numérique $2^{16} - 1$). Notons qu'une procédure peut également rendre plusieurs valeurs: ceci se fait en général en passant des adresses, la fonction modifie l'objet adressé. Si x est un objet, on notera $\&x$ son adresse, et si y est une adresse, y_0 est l'objet modifié (équivalence des tableaux et des pointeurs).

Finalement, une fonction est une procédure C, mais utilisable depuis Lisp. Si le nom Lisp est *foo*, le nom C est en général *Lfoo*, ceci pour éviter des confusions entre la fonction *Lsin*, qui est la fonction qui rend le sinus d'un nombre quelconque et la primitive C qui ne fonctionne que sur

les flottants. Les tirets dans les noms de fonctions sont remplacés par des soulignés. Dans certains cas, comme le cas de +, un nom adéquat est choisi.

Tous les algorithmes sont indexés. On utilise la police **bold** pour référencer la page où est définie une procédure, et une police normale pour référencer toute utilisation. Certaines procédures, comme `iadd1` sont utilisées uniquement dans l'algorithme où elles sont définies. En fait, `iadd1` n'est utilisée qu'une seule fois, la ligne qui précède sa définition. C'est pour ceci que son utilisation n'est pas indexée. L'algorithme d'indexation est le suivant : si A est une procédure qui appelle une procédure B dans un algorithme C, si B n'est pas définie dans C, B sera indexée à la page où A est déclarée. Par exemple, un des algorithmes contient un certain nombre de primitives sur les entiers et les fractions, dont `qx0`. Étant donné a et b cette fonction rend la fraction a/b, en la réduisant s'il le faut. Elle fait donc appel à la fonction `npgcd`. En fait, l'extraction du pgcd se fait par la macro `pgcd_hack`. Pour des raisons de commodités, l'utilisation de `npgcd` est référencée uniquement dans `pgcd_hack`, pas dans les fonctions qui utilisent cette macro. Il se trouve – dans une version provisoire de ce document – que `qx0` est sur la même page que `pgcd_hack`, mais l'appel à `pgcd_hack` dans `qx0` est sur la page d'après.

L'appel au primitives n'est en général pas indexé. Toute référence à `make_int` l'est, dans la mesure où cette procédure alloue de la mémoire. Nous avons également indexé la fonction inverse `Int_val`. Les macros `get_digit`, `set_digit`, `VREF`, `VSET` ne sont pas indexées. En fait, si X est un vecteur de pointeurs ou de chiffres, nous écrirons $X_i = a$ ou $a = X_i$ au lieu de `VSET`(X, i, a) ou $a =$ `get_digit`(X, i). Le type de X étant donné explicitement, le lecteur trouvera facilement laquelle des macros est utilisée. De même, si Y est le pointeur de tas de X, on écrira Y_i pour désigner le i^{e} élément de Y. On le notera parfois $Y[i]$, surtout si i est une variable avec un indice.

On écrira parfois : « $X_i = t/2^p + c, c = t.2^{32-p}$ (multiplication, division par décalage) ». Le code C correspondant est `X[i] = (t>>p)|c; c = t<<(32-p);`. Les appels aux primitives C, comme le décalage ne sont jamais indexés. La plupart des variables globales sont indexées. On omettra les indices de tableau comme i_A, la variable globale `C`, et la variable `mod`.

Remarque finale : dans la mesure du possible, dans un algorithme donné, une variable x a toujours le même type. Il y a trop de types possibles pour pouvoir réserver le même type dans tout le document. On s'est efforcé cependant de respecter les conventions suivantes : i, j, k, l sont des indices, s, t des tailles de vecteurs, X, Y, Z des vecteurs (parfois des pointeurs de tas), A, B, C, D des chiffres. Dans certains cas, nous avons voulu donner une sémantique aux lettres : le quotient et le reste d'une division seront notés q et r, le numérateur et dénominateur d'une fraction seront notés n, d, etc.

1.5 Types

Les objets Lisp sont définis par le code C suivant :

```
typedef union lispunion* p_object;
union lispunion {
   struct a_vector Vector;
   struct a_uvector UVector;
   struct {int type; int D} Int;
   struct {int type; double D} Double;
   ...
```

```
};
struct a_uvector {
   int type;
   int size;
   p_object ll_type;
   unsigned int* h_pointer;
};
struct a_vector {
   int type;
   int size;
   p_object ll_type;
   p_object h_pointer[1];
};
```

Nous n'avons pas indiqué la structure des autres objets comme les symboles, les listes, etc. Chacun de ces objets commence par un champ `type` qui indique le type de l'objet. Le champ `ll_type` d'un vecteur est utilisé pour par exemple construire un système objet au-dessus de Lisp. Nous l'utiliserons pour différencier les entiers des rationnels. Le champ `ll_type` d'un vecteur de bits n'est pas utilisé pour l'instant. Une des idées possibles est d'étendre le système, en autorisant n'importe quel pointeur dans le champ `h_pointer` ; c'est le type qui dira comment interpréter la valeur. Par défaut, l'imprimeur considère qu'il y a `size` mots, qui sont imprimés en base 16 en tant que suite de bits.

Dans les algorithmes qui suivent, nous utiliserons les noms suivants pour désigner les types.

- chiffre : entier 32bits non signé, `(unsigned int)` ;
- tableau de chiffres : tableau C, `(unsigned int*)` ;
- pointeur de tas : tableau de chiffres ;
- petit entier : objet Lisp, de type `Int`, dont la valeur `D` est un entier C, dans l'intervalle $[-2^{32}+1, 2^{32}-1]$;
- petit entier non signé : petit entier dont la valeur est considérée non signée ;
- flottant : objet Lisp, de type `Double`, dont la valeur `D` est un nombre réel, double précision ;
- vecteur de chiffres : objet Lisp, de type `UVector`, contenant un type, une taille, et un tableau de chiffres ;
- vecteur de pointeurs : objet Lisp, de type `Vector`, contenant un type, une taille, et un certain nombre d'objets Lisp ;
- vecteur : vecteur de pointeurs ;
- entier interne : petit entier non signé ou vecteur de chiffres ;
- entier interne normalisé : entier interne satisfaisant des contraintes expliquées dans le chapitre suivant ;
- grand entier : vecteur, de type `#:r:z`, de taille 2, contenant dans le champ `rz_num` un entier interne, la valeur absolue, et dans `rz_sign` le signe sous forme d'un booléen (vrai si positif) ;

- fraction: vecteur, de type `#:r:q`, de taille 4, contenant dans les champs `rz_num` et `rz_den` des entiers internes, le numérateur et dénominateur de la fraction, dans le champ `rq_rflag` un indicateur, qui s'il est vrai, dit que numérateur et dénominateur sont premiers entre eux, et un dernier champ `rq_sign` contenant le signe;
- complexe: vecteur, de type `#:r:c`, de taille 2, contenant dans les champs `rc_rpart` et `rc_ipart` deux nombres réels, les parties réelles et imaginaires de nombre complexe;
- nombre: petit entier, grand entier, flottant, fraction ou complexe;
- liste: objet Lisp de type `cons`, comprenant deux champs: le premier élément de la liste et le reste de la liste. La list vide est représentée par un objet spécial.

Par abus de langage, si x est un entier interne, au lieu de dire: « si x est de type petit entier non signé alors ... sinon ... », on dira: « si x est un chiffre alors, sinon ». Ceci est justifié dans la mesure où dans le premier cas on utilisera toujours `Int_val`(x), qui lui est un chiffre.

1.6 Les primitives Lisp à notre disposition

Note: ce paragraphe est une copie de la version précédente de ce rapport, [2].

Les algorithmes que nous allons présenter dans le premier chapitre sont pour la plupart décrits dans [8, section 4.3] et adaptés à SISYPHE. On suppose savoir manipuler des entiers en base $E = e^p$. Dans la version courante de SISYPHE, on suppose $e = 2$ et $p = 16$. En principe ces valeurs particulières ne devraient pas avoir d'influence sur les algorithmes, mais ceci n'est pas toujours exact. C'est pour cela que nous préciserons chaque fois que cela est nécessaire l'influence de l'exposant (en ce qui concerne la base e, on suppose qu'elle est toujours égale à 2).

Précisons ce qu'on entend par savoir manipuler des entiers en base E. D'une part, on utilise une variable globale `C` (pour « carry », c'est la retenue courante, en interne, elle s'appelle `#:ex:regret`). Le fait d'utiliser une variable globale est important dans la mesure où une fonction ne peut rendre qu'une seule valeur. On préfère modifier `C` et rendre une valeur plutôt que de rendre une liste (les fonctions utilisées ici sont des fonctions de base, on ne peut se permettre d'allouer de la mémoire de façon injustifiée).

On dispose des cinq fonctions suivantes. D'une part, la fonction `ex+` prend deux arguments a et b; elle calcule la somme $a + b +$ `C` sous la forme $CE + d$. La quantité C est mise dans `C`, le résultat de la fonction est d. Notons que $a + b +$ `C` $\leq 3E - 3 < E^2$. La fonction `ex*` prend trois arguments a, b et c. Elle calcule $ab + c +$ `C` sous la forme $xE + y$. Elle rend y, et positionne x dans `C`. Notons que $ab + c +$ `C` $\leq (E-1)^2 + 2(E-1) = E^2 - 1$.

La fonction `ex/` prend deux arguments a et b. Elle réalise la division euclidienne `C`$E + a = xb + y$. Elle positionne y dans `C` et rend x. On suppose `C` $< b$ (en particulier $b > 0$), d'où $x < E$. Comme c'est une fonction de bas niveau, aucun test n'est fait, n'importe quoi peut se produire si cette condition n'est pas satisfaite.

La fonction `ex-` prend un argument a, elle calcule $E - 1 - a$. Définissons la fonction `ex--`(a, b) par `ex+`$(a$,`ex-`$(b))$. Elle calcule donc $E + a - b +$ `C` $- 1$. Dans le cas `C` $= 1$, si $a \geq b$, elle rend donc $a - b$ et laisse `C` à 1. La quantité `C` est appelée retenue dans le cas de l'addition et de la multiplication. Dans le cas de la soustraction, la quantité $1 -$ `C` est appelée emprunt. Finalement on dispose d'une fonction `ex?` qui compare deux arguments a et b. Elle rend 0 si $a = b$, 1 si $a > b$ et $E - 1$ si $a < b$.

1.7 Implémentation des primitives en C

Dans la version C, toutes les primitives manipulent des chiffres, entiers 32bits non signés. Ces primitives ne sont pas accessibles depuis Lisp, de même que la variable qui contient la retenue courante `carry`. Dans certains cas, la retenue courante est passée en argument et est la valeur de retour de la fonction. Dans ce cas, l'un des arguments de la fonction est un pointeur sur le résultat. Les fonctionnalités `ex-` et `ex?` sont écrites sous forme de macro.

Les trois fonctions non triviales sont donc `ex+`, `ex*` et `ex/`. Nous supposons savoir additionner, multiplier et diviser deux nombres de 32bits, avec résultat modulo 2^{32}. Si le compilateur et le système d'exploitation le permettent, on utilise une multiplication et une division 64bits. Dans ce cas, le code des primitives devient trivial. Dans le cas contraire, on utilise les algorithmes qui suivent.

Rappel: E est la base courante, un chiffre est un nombre x avec $0 \leq x < E$. Pour écrire $a + b = CE + x$, on calcule $x = a + b$ modulo E, et on compare le résultat à a: si $x < a$, on a $C = 1$, sinon $C = 0$. De même, pour écrire $a + b + c = C'E + y$, on pose $y = x + c$. Si $x < b$ ou si $y < c$, on a $C' = 1$, sinon $C' = 0$. La fonction `ex_add_c` prend en argument l'adresse de a, les valeurs b et c. Elle rend C' et range le résultat y dans a. C'est une extension commode de `ex+`. Dans le cas particulier $b = 0$ et $c = 1$ (propagation de la retenue), il suffit de calculer $y = a + 1$, et de tester $y < 1$, donc $y = 0$. Le code de `iadd2`, qui utilisait `ex+` devient plus simple (à un tel point qu'on en a fait une macro).

En ce qui concerne la fonction `ex*`, la partie non triviale est le calcul du produit ab. Pour ceci, on pose $E' = 2^{16}$, donc $E = E'^2$, et on écrit $a = uE' + v$, et $b = u'E' + v'$. Alors

$$ab = uu'E'^2 + vv' + (uv' + u'v)E'.$$

On écrit ceci sous la forme $xE + y$. On commence par écrire $x = uu'$ et $y = vv'$. Il s'agit ensuite d'ajouter $(uv' + u'v)E'$. Il y a en principe 4 multiplications. Si on remarque que $uv' + u'v = (u + v)(u' + v') - uu' - vv'$, il suffit de 3 multiplications. Ceci est appelé algorithme de Karatsuba, et sera utilisé plus loin, pour la multiplication des grands nombres.

Algorithme 1. (Multiplication interne) *Toutes les variables locales et paramètres sont des chiffres, sauf X qui est l'adresse d'un chiffre, ou un tableau de chiffres. Toutes les opérations sont modulo 2^{32}. On a $E = 2^{32}$, $E' = 2^{16}$.*

Macro update_product. *Paramètres x, y et c.* [Incrémenter $xE + y$ de c.]

Copier c dans une variable t.

Incrémenter y de t.

Si $y < t$, incrémenter x de 1.

Macro HIBITS. *Paramètre x.* [Si $x = aE' + b$, rend a.]

Rendre x décalé de 16 bits vers la droite.

Macro LOBITS. *Paramètre x.* [Si $x = aE' + b$, rend b.]

Rendre $x \wedge (2^{16} - 1)$ [*et* logique].

Procédure ex_mul_c. *Paramètres a, b, X et c.* [Une implémentation efficace consiste à dupliquer le code de `ex*`, il n'y a pas besoin de modifier `C`.]

Sauver la variable `C` *dans une variable locale. Y mettre c.*

Appeler `ex*`(a, b, X_0).

Mettre le résultat dans X_0, rendre la valeur de `C`*, et remettre* `C` *à sa valeur sauvée plus haut.*

Procédure ex*. *Paramètres a, b et c.* [Calcule $ab + c + \texttt{C} = xE + y$; met x dans `C` et rend y]

Écrire $a = uE' + v$, $b = u'E' + v'$ en utilisant `HIBITS` *et* `LOBITS`.

Poser $x = uu'$, $y = vv'$.

Poser $y' = x + y$. Si $y' < y$, décrémenter x de E'.

Décrémenter x de `HIBITS`(y').

Poser $b' = y'E'$ par décalage, $y' = y$.

Décrémenter y de b'. Si $y > y'$ décrémenter x de 1. [On a $uu'E + vv' - (uu' + vv')E'$ dans $xE + y$.]

Poser $a' = u + v$, $b' = u' + v'$.

Si $a' \geq E'$, incrémenter x de b', et décrémenter a' de E'.

Si $b' \geq E'$, incrémenter x de a', et décrémenter b' de E'.

Appeler `update_product` *sur x, y, c, puis sur $x, y,$* `C`.

Mettre x dans `C`*, et rendre y.*

Macro shift_left. *Paramètres: X le vecteur à décaler, f, l premiers et derniers indices, c, retenue et p taille du décalage.*

Poser $c = 0$.

Pour $i = l, l-1, \ldots, f$ faire : $t = X_i$, $X_i = t2^p + c$, $c = t/2^{32-p}$ [multiplication, division par décalage].

L'algorithme de division sera expliqué plus loin. L'idée essentielle est de se ramener à une division de $aE + b$ par c, où le premier bit de c est positionné. Ensuite, on se ramène à deux divisions 48bits par 32bits. Le quotient estimé de ces division s'obtient en divisant les 32 premiers bits par les 16 premiers bits. Il y a ensuite une correction à faire.

Algorithme 2. (Division interne) *Toutes les variables locales et paramètres sont des chiffres, sauf X qui est l'adresse d'un chiffre, ou un tableau de chiffres. Toutes les opérations sont modulo 2^{32}.*

Macro shift_right. *Paramètres: X le vecteur à décaler, f, l premiers et derniers indices, c, retenue et p taille du décalage.*

Poser $c = 0$.

Pour $i = f$ jusqu'à $i = l$ faire : $t = X_i$, $X_i = t/2^p + c$, $c = t.2^{32-p}$ [multiplication, division par décalage].

Procédure ex/. *Paramètres b et c.* [Divise $\texttt{C}E+b$ par c ; met le reste dans `C` et rend le quotient].

Poser $a = \texttt{C}$.

Si $a = 0$, diviser b par c, mettre le reste dans `C` *et rendre le quotient.*

Si $c \geq 2^{31}$, rendre $\texttt{ex\_quo\_i}(a, b, c, \texttt{LOBITS}(c), \texttt{HIBITS}(c))$.

Si $c = 0$, rendre 0. [C'est une erreur.]

Poser $p = \texttt{anormalise}(c)$.

Poser $t = b/2^{32-p}$, multiplier b et c par 2^p, poser $a = a2^p + t$ [multiplications, divisions via décalage].

Poser $a = \texttt{ex\_quo\_i}(a, b, c, \texttt{LOBITS}(c), \texttt{HIBITS}(c))$.

Diviser `C` *par 2^p.*

Rendre a.

Procédure ex_quo_i. *Paramètres a, b, c, c_1 et c_0.* [On suppose $c = c_0E' + c_1$, $E'/2 \geq c_0 < E$; divise $aE + b$ par c, met le reste dans `C` et rend le quotient].

1. *Poser $u_2 = \texttt{HIBITS}(b)$.*

2. *Poser $q = a/c_0$, $A = qc_1$.*

3. *Poser $B = u_1 2^{16} + u_2$ où u_1 est le reste de la division de a par c_0.*

4. *Poser $r = B - A$.*

5. *Si $A > B$: décrémenter q, ajouter c à r, tant que $r \geq c$.*

6. *Sauver q dans q'. Poser $u_2 = \texttt{LOBITS}(b)$.*

7. *Refaire les points* **2**, **3**, **4** *et* **5**.

8. *Mettre r dans* `C`*. Rendre $q'2^{16} + q$.*

Procédure anormalise. *Paramètre x.*

Poser $t = 2^{31}$, $p = 0$.

Tant que $t \wedge p$ est non nul, diviser t par 2, incrémenter p.

Rendre p.

Chapitre 2

Arithmétique en précision arbitraire

2.1 Introduction

Les algorithmes décrits ici manipulent des entiers positifs, appelés nombres internes, qui sont les briques de base pour implémenter les grands entiers et nombres rationnels (chapitres suivants), ou les bigFloat et les polynômes de SISYPHE. Ces entiers sont représentés en base $E = 2^p$ où $p = 32$. La plupart des algorithmes sont indépendants de la valeur de p.

Un entier entre 0 et $E - 1$ sera appelé un chiffre. Un tableau de chiffres sera aussi appelé pointeur de tas. Un vecteur de chiffres est un vecteur Lisp, de type vecteur de bits, qui comprend essentiellement un type (pour le distinguer des vecteurs de pointeurs), une taille et un pointeur de tas. Si le vecteur est $[a_n, a_{n-1}, \ldots, a_0]$, on lui associe une valeur, le nombre $\sum a_i E^i$. Dans certains cas, la valeur n'occupe pas le vecteur en entier. Si le vecteur est $[b_0, \ldots, b_k, a_n, \ldots, a_0, c_0, \ldots, c_q]$, et la valeur est $\sum a_i E^i$, on dira dans ce cas que a_n est le premier chiffre utile et a_0 est le dernier chiffre utile. Ces deux chiffres sont repérés par deux indices i_A et t_A (i_A est l'indice du premier chiffre utile et t_A est l'indice du dernier chiffre utile). On utilise huit variables globales pour repérer les indices utiles, à savoir i_A, i_B, i_C, i_D, i_E, t_A, t_B et t_C (dans le code C, ces variables s'appellent `index_A` ou `size_A`, etc.). Les indices commencent à 0, de telle sorte que si a occupe le vecteur A en entier, donc si $A = [a_n, \ldots, a_0]$, on a $i_A = 0$ et $t_A = n$, et le vecteur A est de taille $n + 1$. Dans ce cas, si $n > 0$ et a_n est non nul, le nombre a sera dit normalisé. Nous noterons $t(A)$ la taille d'un vecteur A. Les fonctions externes ne font aucune hypothèse sur les variables i_A, etc, ce sont uniquement les fonctions internes qui supposent que certaines conditions sont satisfaites (interne signifie : utilisable uniquement par les algorithmes définis dans ce chapitre).

Un petit entier Lisp X est un objet typé dont la valeur x est un entier signé $-E/2 < x < E/2$. Si on considère X comme entier non signé, on obtient ce qu'on appelera (par abus de langage) un chiffre. Notons qu'il y a unicité des nombres 0 et 1. Notons que parmi les E valeurs possibles, on en exclut une. Suivant les architectures, ceci peut être $-E/2$ ou -0. Avec notre choix, l'opposé d'un petit entier est toujours un petit entier.

Un nombre interne sera soit un chiffre, soit un vecteur de chiffres normalisé. Toutes les fonctions externes prennent en argument des nombres internes, et rendent un nombre interne. Ces fonctions

ne font pas d'hypothèses sur les variables globales, et en général après leur exécution, l'état des variables globales est quelconque. Exception : l'algorithme de division positionne son reste dans la variable `ex_mod`. L'algorithme de Bezout prend en argument a et b, et résoud $Ua + Vb = p$. Le résultat est p, la valeur absolue de U et son signe sont dans deux variables globales.

Toutes les fonctions internes commencent par la lettre i, certaines fonctions auxiliaires commencent par la lettre a. Toutes les fonctions externes commencent par la lettre n.

2.2 Fonctions auxiliaires

Algorithme 3. (Fonctions auxiliaires) *On utilise x, y des vecteurs de chiffres, n, m, t des indices, X, Y des tableaux de chiffres. Rappelons que $t(x)$ est la taille de x. La fonction* `noverflow` *est la seule qui utilise une variable globale, à savoir* `C`.

Macro noverflow1. *Argument x.*

Rendre `noverflow`$(x, t(x))$.

Procédure noverflow. *Paramètres x et n (un vecteur de chiffres et un indice).*

Si `C` $= 0$, *rendre x si $n \neq 1$,* `make_int`(x_0) *sinon.*

Poser $y =$ `acopyvector3`$(x, n+1, 1)$. *Mettre* `C` *dans y_0 et rendre y.*

Procédure nunderflow1. *Paramètre x.*

Rendre `nunderflow3`$(x, 0, t(x) - 1)$.

Procédure nunderflow3. *Paramètres x, n, t.*

Si $x_n = 0$, si $n = t$ rendre 0, sinon `nunderflow3`$(x, n+1, t)$.

Sinon, rendre `nunderflow2`$(x, n, t+1)$.

Procédure nunderflow2. *Paramètres x, n, t.*

Si $n + 1 = t$, rendre `make_int`(x_n).

Si $n = 0$ et $t = t(x)$ rendre x.

Sinon, rendre `acopyvector4`$(x, n, t - n)$.

Procédure nunderflow4. *Paramètres x, n, t.*

Si $n > t$, rendre 0.

Sinon, rendre `nunderflow3`(x, n, t).

Procédure fillvector. *Paramètres X, n, t.*

Mettre t zéros dans X à partir de n.

Procédure bltvector. *Paramètres x, n, y, m.*

Poser $t = \min(t(x) - n, t(y) - m)$.

Appeler `ibltvector`$($`UHEAP`$(x), n,$ `UHEAP`$(y), m, t)$.

Procédure ibltvector. *Paramètres X, n, Y, m, t.*

Pour i de 0 à $t-1$, remplacer X_{i+n} par Y_{i+m}.

Macro acopyvector1. *Paramètre x. Rendre* `acopyvector5`$(x, t(x), 0, 0)$.

Macro acopyvector2. *Paramètres x, n. Rendre* `acopyvector5`$(x, n, 0, 0)$.

Macro acopyvector3. *Paramètres x, n, m. Rendre* `acopyvector5`$(x, n, m, 0)$.

Macro acopyvector4. *Paramètres x, n, t. Rendre* `acopyvector5`$(x, n, 0, t)$.

Procédure acopyvector5. *Paramètres x, n, m, t.*

Allouer un vecteur y de taille n.

Soient X et Y les pointeurs de tas de x et y.

Soit s le minimum entre $t(x)-t$ et $n-m$.

Pour i de 0 à $s-1$, remplacer Y_{i+m} par X_{i+t}.

Rendre y.

Procédure afind0_beg. *Paramètres X, i, t.*

Tant que $i \leq t$: si X_i est non nul, rendre i, sinon incrémenter i.

Rendre i [i.e., $t+1$].

Procédure afind0_end. *Paramètres X, t, i.*

Tant que $i \geq t$: si X_i est non nul, rendre i, sinon décrémenter i.

Rendre i [i.e., $t-1$].

2.3 Addition et Soustraction

Le code de l'addition et de la soustraction est relativement simple. Nous ne le commenterons donc pas.

Algorithme 4. (Addition des entiers) *Les variables locales et paramètres utilisés sont x et y des entiers internes, a, b des chiffres, h, h' des pointeurs de tas et i, j, n et s, des indices.*

Procédure nadd, addition externe. *Paramètres x et y* [cette fonction rend la somme des deux nombres internes sous forme d'un nombre interne].

Si x et y sont des chiffres, poser $a =$ `Int_val`(x), $b =$ `Int_val`(y). *Poser $b = a+b$. Si $b < a$, rendre le vecteur $[1, b]$, sinon* `make_int`(b).

Si x et y sont des vecteurs de chiffres, appeler `iadd1`(x, y), *sinon* `iadd3`(x, y).

Finir en appelant `iadd2`.

Macro iadd1. *Paramètres x, y* [positionne c, i, n, x et h. Après l'appel, il faut ajouter c, qui est 0 ou 1 à la quantité x en position i, propagation de la retenue. La macro additionne deux vecteurs de chiffres].

Si y est plus long que x, échanger x et y.

Copier x.

Soit $n = t(x)$, $j = t(y) - 1$, h le pointeur de tas de x et h' celui de y.

Poser $c = 0$, $i = n - 1$. Tant que $j \geq 0$, poser $c =$ `ex_add_c`$(\&h_i, h'_j, c)$ *et décrémenter i et j.*

Macro iadd3. *Paramètres x et y* [même sémantique que la macro précédente, l'un des arguments est un chiffre, l'autre un vecteur de chiffres].

Si x est un chiffre, poser $c =$ `Int_val`(x) *et $x = y$, sinon $c =$* `Int_val`(y).

Copier x.

Soit $n = t(x)$, h le pointeur de tas de x.

Poser $c =$ `ex_add_c`$(\&h_{n-1}, c, 0)$ *et $i = n - 2$.*

Macro iadd2. *Paramètres c, i, n, x et h.* [n taille de x, propager la retenue à partir de i].

Si $c = 0$, rendre x.

Tant que $i \geq 0$ et h_i incrémenté de 1 est nul, décrémenter i.

Si $i \geq 0$, rendre x.

Sinon, copier x via `acopyvector3`$(x, n + 1, 1)$, *poser $x_0 = 1$, rendre x.*

On va maintenant définir la soustraction. On suppose qu'elle a un sens, donc que $x \geq y$. En particulier la taille de x est au moins la taille de y.

Algorithme 5. (Soustraction des entiers) *Les variables et paramètres utilisés sont x et y des entiers internes, X et Y des vecteurs de chiffres, i, j, et k, des indices, h, h' des pointeurs de tas.*

Procédure ndiff, soustraction externe. *Paramètres x et y* [rend la différence $x - y$].

Poser `C` $= 1$.

Si x est un chiffre, rendre `make_int`(`ex--`(`Int_val`(x), `Int_val`(y))).

Si y est un chiffre, soit $i = t(x) - 1$. Copier x, poser $h =$ `UHEAP`(x). *Remplacer h_i par* `ex--`$(h_i,$ `Int_val`$(y))$. *Rendre* `idiff2`$(x, i - 1, h)$.

Sinon, copier x, rendre `idiff1`$(x, y, t(x) - 1)$.

Procédure idiff1. *Paramètres X, Y et i* [$i + 1$ est la taille de X, suppose `C` $= 1$].

Soient $j + 1$ la taille de Y, h et h' les pointeurs de tas de X et Y.

Tant que $j \geq 0$, remplacer h_i par `ex--`(h_i, h'_j), *décrémenter i et j.*

Rendre `idiff2`(X, i, h).

Procédure idiff2. *Paramètres X, i, h* [propager l'emprunt à partir de i].

Si `C` *est nul : Tant que $i \geq 0$: décrémenter h_i, si h_i était nul, décrémenter i, et continuer la boucle.*

Rendre `nunderflow1`(X).

2.4 Multiplication

Nous allons proposer ici l'algorithme classique de multiplication. Il est un peu long car on teste plusieurs cas particuliers, et on optimise les multiplications par les puissances de E, par 0, par 1, etc.

Algorithme 6. (Multiplication des entiers) *Les paramètres et variables locales utilisés sont x et y, des nombres internes, X, Y, z et Z, des vecteurs de chiffres, X', Y' et Z', des tableaux de chiffres, a, b et c des chiffres, et i, j, k, k_0, t_1, t_2, s_1 et s_2, des indices.*

Procédure ntimes, multiplication externe. *Paramètres x et y* [rend le produit xy].

Si x et y sont des chiffres, rendre `ntimesfix`$(\texttt{Int\_val}(x), \texttt{Int\_val}(y), 0)$.

Si x est un chiffre, rendre `ntimesfix1`$(\texttt{Int\_val}(x), y)$.

Si y est un chiffre, rendre `ntimesfix1`$(\texttt{Int\_val}(y), x)$.

Sinon, soient t_1+1 et t_2+1 les tailles de x et y. De plus soit $s_1 =$ `afind0_end`$(\texttt{UHEAP}(x), 0, t_1)$, *et s_2 la quantité correspondante pour y.*

Si $s_1 = s_2 = 0$, rendre `ntimesfix`$(x_0, y_0, t_1 + t_2)$.

Si $s_1 = 0$, poser $t_A = t_2 + 1$ et $t_B = s_2$, et rendre `ntimesc1`(y, x_0, t_1).

Si $s_2 = 0$, poser $t_A = t_1 + 1$ et $t_B = s_1$, et rendre `ntimesc1`(x, y_0, t_2).

Soit $l = \min(1 + s_1, 1 + s_2)$, $k = s_1 + s_2 + 2$ et $L = t_1 + t_2 + 2$.

Si $l \geq K$ (K flag de Karatsuba), allouer un vecteur p de taille $2l$, sinon y mettre un vecteur constant de taille 0.

Allouer un vecteur z de taille L.

Appeler `itimes2` *sur les pointeurs de tas de x, y, z, p, et les entiers 0, s_1, 0, s_2, 0 et $k - 1$.*

Rendre `nunderflow1`(z).

Procédure ntimesfix. *Paramètres a, b et i* [on calcule abE^i].

Poser `C` $= 0$, $c =$ `ex*`$(a, b, 0)$.

Si `C` $\neq 0$, *allouer un vecteur de taille $i + 2$. Y mettre* `C` *en position 0, et c en position 1, et rendre ce vecteur.*

Si $i = 0$, rendre `make_int`(c).

Sinon, allouer un vecteur de taille $i + 1$, y mettre c en position 0, et le rendre.

Procédure ntimesfix1. *Paramètres a et Y.*

Si $a = 0$, rendre 0.

Si $a = 1$, rendre Y.

Sinon, soit $t_A = t(y)$, $t_B =$ `afind0_end`$(\texttt{UHEAP}(y), 0, t_A - 1)$.

Rendre `ntimesc1`$(Y, a, 0)$.

Procédure ntimesc1. *Paramètres X, a et i* [calculer aXE^i ; X est de taille t_A, normalisé, il y a des 0 après t_B].

Si $a = 1$, et $i = 0$, rendre X.

Sinon, remplacer X *par* `acopyvector2`$(X, i + t_A)$.

Si $a = 1$, *rendre* X.

Sinon, positionner i_B *à 0, calculer* `itimesc0`$($`UHEAP`$(X), a, 0)$. *Rendre* `noverflow1`(X).

Procédure itimesc0. *Paramètres* X', *un tableau de chiffres (partie utile entre* i_B *et* t_B*),* a *et* b. [Calculer $Xa + b$. En cas d'overflow, essaie de décrémenter i_B pour mettre l'overflow dans le vecteur X. Sinon l'overflow sera dans `C`].

Poser `C` $= b$, $i = i_B$ *et* $j = t_B$.

Tant que $i \leq j$, *remplacer* X'_j *par* `ex*`$(X'_j, b, 0)$ *et décrémenter* j.

Si $j < 0$ *ou* `C` $= 0$, *fin.*

Sinon positionner `C` *dans* X'_j, *poser* `C` $= 0$, *et* $i_B = j$.

Procédure itimes1. *Paramètres* X', Y' *et* Z', i_A, s_A, i_B, s_B, i_C *et* s_C. [Il s'agit de calculer $XY + Z$ dans le vecteur Z. Les premiers indices dans les vecteurs sont i_A, i_B et i_C respectivement, et les tailles sont s_A, s_B et s_C. On suppose que $iC = i_A + i_B$. La fonction rend la retenue éventuelle. Elle n'utilise aucune variable globale.].

1. *Poser* $t_B = s_B + i_B - 1$, $t_A = s_A + i_A - 1$, $k_0 = s_C + i_C - 1 - t_B$, $n = k_0 - s_A$ *et* $c' = 0$.

2. *Pour* $j = t_B$, *tant que* $j \geq i_B$

2.1. *Poser* $c = 0$, $A = Y'_j$.

2.2. *Si* $A = 0$, *poser* $k = j + n$.

2.3. *Sinon, poser* $i = t_A$, $k = k_0 + j$. *Tant que* $i \geq i_A$, *remplacer* c *par la quantité* `ex_mul_c`$(X'_i, A, \&Z'_k, c)$, *décrémenter* i *et* k.

2.3. *Remplacer* c' *par* `ex_add_c`$(\&Z'_k, c, c')$.

3. *Rendre* c'.

2.5 Karatsuba

Il s'agit d'une optimisation de l'algorithme de multiplication. Soit à calculer le produit de deux nombres x et y ayant 2^n chiffres en base E. Posons $B = E^{2^{n-1}}$, de sorte que $x = aB + b$ et $y = cB + d$, où a, b, c et d sont des nombres en base B. On a

$$xy = acB^2 + (ad + bc)B + bd\,; \qquad ad + bc = (a + b)(c + d) - ac - bd.$$

L'algorithme classique utilise 4^n multiplications en base E, l'algorithme de Karatsuba récursif n'en utilise que 3^n (3 multiplications en base B au lieu de 4). La vraie complexité est difficile à estimer : il y a des additions et des soustractions supplémentaires, et des propagations de retenues. De plus, si la taille de x n'est pas une puissance de 2, l'algorithme ne s'applique pas directement. En d'autres termes, pour n petit, il vaut mieux éviter cet algorithme. On utilise une variable globale K, qui est la taille minimale de x à partie de laquelle on utilise Karatsuba.

Notons que l'algorithme normal de multiplication `itimes1` est quadratique. Une analyse détaillée montre que le produit de deux nombres de n et m chiffres est $anm + bm + c + \epsilon$, où ϵ tient compte di « si » en 2.2. Celui-ci ajoute à la complexité un terme λn, avec une probabilité de 2^{-32}, ce qui est négligeable. La complexité effective moyenne du produit de deux nombres de

n chiffres est donc quadratique, $an^2 + bn + c$. Nous avons essayé de déterminer ces constantes. Cependant, l'imprécision naturelle des mesures de temps CPU sur machine partagée a été trop grande pour pouvoir calculer ces constantes. L'algorithme de Karatsuba est beaucoup plus compliqué, le nombre de tests est beaucoup plus important, et estimer numériquement quoi que ce soit est encore plus compliqué.

Soit $C(n)$ le nombre de multiplications 32 bits utilisées pour multiplier deux nombres de taille n, en utilisant l'algorithme de Karatsuba. On a

$$C(2n) = 3C(n) \qquad C(2n+1) = 3C(n) + 4n + 1.$$

Il n'y a pas de formules explicites pour $C_K(n)$ pour K et n quelconques en fonction de a, b, c, n et K. Cependant, en admettant $K = 0$, on obtient

$$C(2^n - 1) = \frac{3}{2} - 2^{n+2} + \frac{5}{2}3^n.$$

En admettant que ceci soit le pire des cas, la complexité de l'algorithme est donc $Cn^{\log 3/\log 2}$, où C est entre 1 et 5/2.

Supposons que les deux nombres à multiplier n'aient pas la même taille. Par exemple, x est de taille 8 et y de taille 9. Écrivons que y est 1 chiffre suivi de 8 chiffres. Ceci donne 35 multiplications. Si y est de taille 7, en rajoutant un 0 initial, on a besoin de 27 multiplications. Si par contre on fait comme avant, i.e., on dit que x a $1 + 7$ chiffres, on arrive au total de 50 multiplications (rappel: l'algorithme classique en utilise 56). On pourrait aussi compter le nombre d'additions et de soustractions. Dans ce cas, il faudrait tenir compte du coût relatif entre l'addition et la multiplication, ce qui dépend non seulement de la machine, mais aussi du processeur utilisé. Trouver la meilleure stratégie est non évident.

On utilise la stratégie suivante: on suppose x plus grand que y, $E^{n-1} \le y < E^n$. Si $n < K$, on utilise l'algorithme classique. Sinon, on pose $B = E^n$, et on écrit x et y en base B. Par construction y est un chiffre et $x = \sum x_i E^i$ a $k+1$ chiffres. Pour $i < k$, on calcule $x_i y E^i$, on somme tout cela en propageant la retenue, et il reste à calculer $x_k y E^k$. Si x_k est de taille n, il n'y a pas de problème, sinon on est ramené au cas précédent.

Dans les autres cas, Karatsuba calcule $NM + R$ où N et M sont de taille n, R est de taille $2n$, et on utilise un vecteur auxiliaire P de taille $2n$. Dans le cas où n est impair, on écrit $N = N'E + x$, $M = M'E + y$ où x et y sont des chiffres, N' et M' sont de taille paire. On a

$$NM + R = N'M'E^2 + N'yE + (M'E + y)x + R.$$

Le premier produit s'effectue par un appel à Karatsuba, les autres sont triviaux: multiplication d'un vecteur de chiffres par un chiffre.

On suppose maintenant que N et M sont de taille paire $2n$. Soit $B = E^n$. On pose $N = N_1B + N_0$, $M = M_1B + M_0$, $R = R_0 + R_1B + R_2B^2 + R_3B^3$. On coupe le vecteur de travail P en 4 morceaux, P_0, P_1, P_2 et P_3. Aux étapes 4 et 5, on calcule $N_0 + N_1$ dans P_3 et $M_0 + M_1$ dans P_2. Il s'agit d'additionner $(M_0 + M_1)(N_0 + N_1)B$ à R. On ne modifie que R_1 et R_2, et on garde la retenue pour la suite. S'il y a une retenue pour $N_0 + N_1$, on ajoute $M_0 + M_1$ à R_2, et s'il y a une retenue pour $M_0 + M_1$, on ajoute $N_0 + N_1$ à R_2. À l'étape 8, on ajoute $(M_0 + M_1)(N_0 + N_1)B$ à R, en utilisant P_0P_1 comme espace de travail. La retenue courante est alors un nombre entre 0 et 3.

Une fois ceci fait, il s'agit d'additionner à R la quantité $N_0M_0 + N_1M_1B^2 - (N_0M_0 + N_1M_1)B$. On calcule N_0M_0 dans $P_0 + P_1B$ en utilisant P_2P_3 comme espace de travail. Puis on soustrait ce produit de $R_1 + R_2B$ (à ce moment, la retenue est a priori entre -1 et 4) et on ajoute N_0M_0 à R, en fait à $R_0 + R_1B$, avec propagation de la retenue vers R_2. Un calcul facile montre que la retenue est maintenant entre 0 et 3 (étapes 10 et 11).

Finalement, on calcule N_1M_1, on soustrait ce produit de $R_1 + R_2B$. On ajoute ensuite $N_1M_1B^2$ à R, en procédant en deux temps: on ajoute d'abord n chiffres, puis les n derniers en tenant compte de la retenue à propager. Notons que les calculs de N_0M_0 et N_1M_1 ne peuvent pas provoquer d'overflow.

Algorithme 7. (Multiplication via Karatsuba) *Toutes les variables locales et paramètres utilisés sont des chiffres ou indices sauf N, M, P, et R qui sont des tableaux de chiffres. Les hypothèses faites sur ces vecteurs sont les mêmes et données dans la fonction principale. Dans* `call_karatsuba`, *x, y, z et p sont des vecteurs de chiffres.*

Procédure Karatsuba, fonction principale. *Paramètres N, M, R, P, l, i_A, i_B, i_C et i_D.* [N et M sont de taille l, R et P de taille $2l$. Les premiers indices utiles sont dans i_A, i_B, i_C et i_D respectivement. R est remplacé par $R + MN$, la fonction rend la retenue. Cette fonction n'utilise pas de variables globales, autre que `C` et K, le flag de Karatsuba]

Si $l = 0$, rendre 0.

Si $l = 1$, poser $c =$ `ex_add_c`$(N[i_A], M[i_B], \&R[i_C + 1], 0)$ *et rendre* `ex_add_c`$(\&R[i_C], c, 0)$.

Si $l < K$, rendre `itimes1`$(N, M, R, i_A, l, i_B, l, i_C, 2l)$.

Si l est impair, rendre `karatsuba_odd`$(N, M, R, P, l, i_A, i_B, i_C, i_D)$.

Sinon, rendre `karatsuba_even`$(N, M, R, P, l, i_A, i_B, i_C, i_D)$.

Procédure karatsuba_odd. *Paramètres N, M, R, P, l, i_A, i_B, i_C et i_D.* [Suppose l impair, $l \geq 3$.]

1. *Décrémenter l. Poser $L = 2l$.*

2.a. *Poser $x = M[i_B + l]$, $c = 0$.* [Allons-y pour $N'yE$]

2.b. *Poser $j = i_A + l - 1$, $i = i_C + L$, tant que $j \geq i_A$, poser $c =$* `ex_mul_c`$(N_j, x, \&R_i, c)$, *décrémenter i et j.*

2.c. *Si $c \neq 0$, poser $c =$* `ex_add_c`$(\&R_i, 0, c)$.

2.d. *Poser $c' = c$.*

3.a. *Poser $x = N[i_A + l]$, $c = 0$.* [Allons-y pour $(M'E + y)x$]

3.b. *Poser $j = i_B + l$, $i = i_C + L + 1$, tant que $j \geq i_B$, poser $c =$* `ex_mul_c`$(M_j, x, \&R_i, c)$, *décrémenter i et j.*

3.c. *Si $c \neq 0$, poser $c =$* `ex_add_c`$(\&R_i, 0, c)$.

4.a. *Poser $c' = c + c'$.* [Propagation de la retenue]

4.b. *Poser $i = i_C + l - 1$.*

4..c *Appeler* `prop_carry`(R, i, i_C, c').

5. *Poser $c =$* `Karatsuba`$(N, M, R, P, l, i_A, i_B, i_C, i_D)$.

6. *Rendre $c + c'$.*

Macro karatsuba_add1. *Arguments I, J, K, L et M.* [Calcule $M_0 + M_1$ ou $N_0 + N_1$ dans P. Les 4 premiers arguments sont les derniers indices de M_0, M_1, du résultats (augmentés de 1), et le premier indice de M_0. Le dernier argument est le vecteur M ou N.]

1. *Poser* $\texttt{C} = 0$.

2. *Poser* $i = I - 1$, $j = J - 1$, $k = K - 1$.

3. *Tant que* $i \geq L$, *poser* $P_k = \texttt{ex+}(M_i, M_j)$, *décrémenter* i, j *et* k.

4. *Rendre* `C` *comme valeur de retour de la macro.*

Macro karatsuba_overflow. *Arguments I, J et L* [Dans le cas où c_n, la retenue de la macro précédente, est non nulle, on ajoute au résultat partiel l'autre terme; I et J sont un de plus que le dernier indice du terme à ajouter et du résultat et L la dernière valeur de i.]

1. *Poser* $c = 0$.

2. *Poser* $i = I - 1$, $j = J - 1$.

3. *Tant que* $i \geq L$ *poser* $c = \texttt{ex\_add\_c}(\&R_j, P_i, c)$, *décrémenter* i *et* j.

4. *Rendre* c *comme valeur de retour de la macro.*

Macro karatsuba_add2. *Arguments I, J et L.* [Soustrait $N_0 M_0$ ou $N_1 M_1$ de R. Les arguments sont un de plus que les indices de fin du vecteur à ajouter et du résultat, et le premier indice du vecteur à ajouter.]

1. *Poser* $\texttt{C} = 1$.

2. *Poser* $i = I - 1$, $j = J - 1$.

3. *Tant que* $i \geq L$, *poser* $R_k = \texttt{ex--}(R_k, P_i)$, *décrémenter* i *et* j.

4. *Rendre* $\texttt{C} - 1$ *comme valeur de retour de la macro.*

Procédure karatsuba_even. *Paramètres N, M, R, P et l.*

1. *Diviser l par 2.*

2. *Poser* $n_0 = i_A$, $n_1 = n_0 + l$, $n_2 = n_1 + l$. *Définir de même* m_0, m_1, m_2, r_0, r_1, r_2, r_3, r_4, p_0, p_1, p_2, p_3 *et* p_4.

3. *Appeler* $\texttt{fillvector}(P, p_0, 4l)$.

4. *Poser* $c_n = \texttt{karatsuba\_add1}(n_1, n_2, p_4, n_0, N)$.

5. *Poser* $c_m = \texttt{karatsuba\_add1}(m_1, m_2, p_3, m_0, M)$.

6. *Si* c_n *est non nul, poser* $c_n = c_m + \texttt{karatsuba\_overflow}(p_3, r_2, p_2)$.

7. *Si* c_m *est non nul, poser* $c_n = c_n + \texttt{karatsuba\_overflow}(p_4, r_2, p_3)$.

8. *Incrémenter* c_n *de* $\texttt{Karatsuba}(P, P, R, P, l, p_2, p_3, r_1, p_0)$.

9. *Appeler* $\texttt{fillvector}(P, p_0, 2l)$, *et* $\texttt{Karatsuba}(N, M, P, P, l, n_1, m_1, p_0, p_2)$.

10. *Incrémenter* c_n *de* $\texttt{karatsuba\_add2}(p_2, r_3, p_0)$.

11.1. *Poser* $c = 0$.

11.2. *Pour* $i = p_2 - 1$, $j = r_4 - 1$, *tant que* $i \geq p_0$, *poser* $c = \texttt{ex\_add\_c}(\&R_j, P_i, c)$, *décrémenter* i *et* j.

11.3. *Appeler* $\texttt{prop\_carry}(R, j, r_1, c)$.

11.4. *Incrémenter* c_n *de* c.

12. *Appeler* `fillvector`$(P, p_0, 2l)$, *et* `Karatsuba`$(N, M, P, P, l, n_0, m_0, p_0, p_2)$.

13. *Incrémenter* c_n *de* `karatsuba_add2`(p_2, r_3, p_0).

14.1. *Poser* $c = 0$.

14.2. *Pour* $i = p_2 - 1$, $j = r_2 - 1$, *tant que* $i \geq p_1$, *poser* $c =$ `ex_add_c`$(\&R_j, P_i, c)$, *décrémenter* i *et* j.

14.3. *Incrémenter* c *de* c_n.

14.4. *Tant que tant que* $i \geq p_0$, *poser* $c =$ `ex_add_c`$(\&R_j, P_i, c)$, *décrémenter* i *et* j.

15. *Rendre* c.

Procédure itimes0. *Paramètres* N, M, R, P. [Même sémantique que la fonction `itimes2` qui suit, sauf que les tailles sont dans des variables globales. La procédure calcule i_C l'indice du premier chiffre non nul du résultat. La fonction suppose $i_C \geq 0$, c'est une erreur fatale sinon. Pour des raisons historiques i_B et t_B sont les indices du premier argument.]

Poser $i_C = t_C + 1 - (t_B - i_B + 1) - (t_A - i_A + 1)$.

Erreur si $i_C < 0$.

Soit $c =$ `itimes2`$(N, M, R, P, i_B, t_B, i_A, t_A, i_C, t_C)$.

Erreur si $c \neq 0$.

Poser $i_C =$ `afind0_beg`(R, i_C, t_C).

Procédure itimes2. *Paramètres* N, M, R, P, i_A, t_A, i_B, t_B, i_C *et* t_C. [Remplace R par $R+MN$, en utilisant le vecteur auxiliaire P (en fait, N, M, R et P sont des pointeurs de tas). Les arguments supplémentaires sont les premiers et derniers indices utiles des vecteurs. On suppose que la taille de R est au moins la taille de N plus la taille de M. La procédure rend la retenue. Dans le cas où le minimum l de la taille de N et la taille de M est plus grand ou égal au flag de Karatsuba, on suppose que P est un vecteur et que les $2l$ premières cases peuvent être modifiées. Dans le cas contraire, P ne sert pas.]

0. *Poser* $c' = 0$.

1. *Poser* $s_1 = t_A - i_A + 1$ *et* $s_2 = t_B - i_B + 1$.

2. *Si* $s_1 = 0$ *ou* $s_2 = 0$, *fin de la procédure, rendre* c'.

3. *Si* $s_1 < s_2$, *échanger* N *et* M, i_A *et* i_B, t_A *et* t_B, s_1 *et* s_2.

4. *Si* $s_2 = 1$ [Produit d'un vecteur et d'un chiffre]

4.1. *Poser* $A = M[i_B]$.

4.2. *Pour* $i = t_A$, $j = t_C$, *tant que* $i \geq i_A$ *poser* $c =$ `ex_mul_c`$(N_i, A, \&R_j, c)$, *décrémenter* i *et* j.

4.3. *Appeler* `prop_carry`(R, j, i_C, c).

4.4. *Rendre* $c + c'$.

5. *Si* $s_2 < K$ *(flag de Karatsuba)*

5.1. *Soit* $c =$ `itimes1`$(N, M, R, i_A, s_1, i_B, s_2, i_C, t_C - i_C + 1)$.

5.2. *Rendre* $c + c'$.

6. *Tant que* $s_1 \geq s_2$

6.1. *Poser* $k = t_C - s_2 + 1$.

6.2. *Poser* $c =$ `Karatsuba`$(N, M, R, P, s_2, t_A - s_2 + 1, i_B, k, 0)$.

6.3. *Décrémenter* k, *appeler* `prop_carry`(R, k, i_C, c).

6.4. *Incrémenter* c' *de* c.

6.5. *Décrémenter* t_A, s_1 *et* t_C *de* s_2.

7. *Repartir en* **1**.

Macro prop_carry. *Paramètres* R, i, l *et* c.

Si $c \geq 2$, *poser* $c =$ `ex_add_c`$(\&R_i, c, 0)$, *décrémenter* i.

Si c *est non nul : tant que* $i \geq l$ *et* R_i *incrémenté est nul, décrémenter* i. *Ensuite, si* $i \geq l$, *poser* $c = 0$ *sinon* $c = 1$.

2.6 Division

Division d'un nombre par un chiffre

Les choses se compliquent un peu, car l'algorithme de division est non trivial. Commençons par un algorithme qui divise un nombre par un chiffre. S'il n'y avait pas l'étape de normalisation, le code serait trivial. Dans le cas où le hard permet une division 64bits par 32bits, cette étape pourrait être supprimée.

Algorithme 8. (Division par un chiffre) *On utilise un nombre interne* x, *un pointeur de tas* h, *un vecteur de chiffres* X, *un chiffre* y *et un indice* i. [Toutes les procédures rendent le quotient et mettent le reste dans la variable `C`].

Procédure nquoc. *Paramètres* x *et* y [suppose $x > y$, y chiffre].

Poser `C` $= 0$.

Si $y = 1$, *rendre* x.

Si x *est un chiffre, rendre* `make_int(ex/(Int_val`$(x), y)$`)`.

Si x *n'a que deux chiffres* x_0 *et* x_1, *et* $x_0 < y$, *poser* `C` $= x_0$, *et rendre* `make_int(ex/`(x_1, y)`)`.

Sinon rendre `nunderflow1(iquoc0`(x, y)`)`.

Procédure iquoc0. *Données* X *et* y [suppose $X > y$].

Poser $i_B = 0$, $t_B = t(X) - 1$.

Copier X.

Calculer `iquoc(UHEAP`$(X), y)$.

Rendre X.

Procédure iquoc. *Données* h *et* y [Diviser h entre i_B et t_B, suppose $i_B < t_B$, i_B indice du premier chiffre du quotient, remis à jour par la procédure].

Poser `C` $= 0$, *et* $i = i_B$.

Si $y = 1$, fin de la procédure.

Poser $p =$ `anormalise`(y).

Si p est non nul, calculer `shift_left`(h, i_B, t_B, c, p), *et mettre c dans* `C`.

Multiplier y par 2^p par décalage, poser $y = y' + y''2^{16}$ en utilisant `LOBITS` *et* `HIBITS`.

Remplacer h_i par `ex_quo_i`$(\mathtt{C}, h_i, y, y', y'')$ *et incrémenter i. Si le quotient h_{i-1} est nul, poser $i_B = i$.*

Tant que $i \leq t_B$, remplacer h_i par `ex_quo_i`$(\mathtt{C}, h_i, y, y', y'')$ *et incrémenter i.*

Diviser `C` *par 2^p par décalage.*

Division de deux grands nombres

Ceci est maintenant le gros algorithme de division. La taille du code est relativement petite, mais la justification est assez longue. Le but est de diviser un nombre u par un nombre v. On exclut les cas triviaux: d'une part on suppose que v a au moins deux chiffres, et d'autre part $u > v > 0$. On va également supposer que v est normalisé.

En entrée, on suppose que la partie utile de u est entre i_A et t_A, celle de v entre i_B et t_B. L'algorithme de division va modifier u entre i_C et t_A, mais ne modifie pas v. Le quotient et le reste sont dans u, le quotient est cadré à gauche, commençant en i_C, le reste sera cadré à droite, se terminant à l'indice t_A. Entre le quotient et le reste, il n'y a que des zéros. On suppose $i_C < i_A$. En effet, si u est de taille $n+1$ et v de taille $m+1$, le reste aura $m+1$ chiffres, ou moins si le premier chiffre est nul, et le quotient aura $n-m$ chiffre, ou $n-m+1$ dans certains cas. À la fin de l'algorithme, les indices sont changés comme suit: le quotient est entre i_A et t_A, le reste entre i_B et t_B. Note: c'est `iquomod4` qui modifie i_A, pas la fonction qui va suivre.

On va maintenant supposer que $v = v_m + v_{m-1}E + \ldots + v_0E^m$, et noter cela par $v = (v_0 v_1 \ldots v_m)$. On suppose que u a $n+1$ chiffres, $u = (u_{-1}u_0u_1 \ldots u_n)$, où $u_{-1} = 0$. À l'itération j on calcule le j^{e} chiffre du quotient, et on divise les $m+2$ chiffres de u entre $j-1$ et $j+m$ par v. Il s'agit de résoudre

$$(u_{j-1}u_j \ldots u_{j+m}) = q_j(v_0v_1 \ldots v_m) + (r_0r_1 \ldots r_m). \tag{1}$$

On remplacera u_{j-1} par q_j, et u_{j+k} par r_k. Dans le cas où le premier chiffre du quotient est nul, les chiffres du quotient seront décalés de 1. Divisons l'équation (1) par E^{m-1}. Il vient:

$$u_{j-1}E^2 + u_jE + u_{j+1} + \alpha = q_j(v_0E + v_1 + \beta) + r_0E + r_1 + \gamma \tag{2}$$

avec

$$0 \leq \alpha < 1 \qquad 0 \leq \beta < 1 \qquad 0 \leq \gamma < 1.$$

Dans le cas de `two_div`, on suppose que v a deux chiffres, donc $\beta = 0$, on en déduit $\alpha = \gamma$. On fait les hypothèses suivantes:

$$u_{j-1}E^2 + u_jE + u_{j+1} + \alpha < E(v_0E + v_1 + \beta) \tag{H_1}$$

$$r_0E + r_1 + \gamma < v_0E + v_1 + \beta \tag{H_2}$$

L'hypothèse de normalisation que nous allons faire entraîne que v_0 est non nul. Comme $u_{-1} = 0$, l'hypothèse H_1 est vraie pour $j = 0$, dans les autres cas elle est une conséquence de H_2 pour $j-1$. Avant de calculer la solution de (2) sous la contrainte H_2, nous allons montrer qu'elle existe et est unique.

Pour cela, faisons varier q entre $-\infty$ et $+\infty$, et considérons $u-vq$. C'est une quantité strictement croissante, il existe donc q tel que $r = u - vq \geq 0$ et $r' = u - v(q+1) < 0$. On a donc $0 \leq r < v$, donc H_2. Il est clair que $q \geq 0$. Maintenant H_1 dit $q < E$.

Cette hypothèse H_1 implique aussi que $u_{j-1} \leq v_0$. Écrivons $u_{j-1}E + u_j = qv_0 + r$ par division. Dans le cas $u_{j-1} < v_0$ on aura $0 \leq q < E$; dans le cas contraire, on posera $q = E$,$r = u_j$. On a

$$rE + u_{j+1} + \alpha = v_0E(q_j - q) + q_j(v_1 + \beta) + r_0E + r_1 + \gamma. \tag{3}$$

On a $q_j \leq q$: en effet le membre de gauche est $< v_0E$, car $u_{j+1} + \alpha < E$, et $r \leq v_0 - 1$. Le membre de droite est au moins $v_0E(q_j - q)$. Dans le cas $u_{j-1} = v_0$, on pose $q = E$, donc $q_j \leq q$ est trivialement vraie.

Notons que q peut être une très mauvaise estimation: si $v_0 = 1$, $v_1 = E - 1$, $u_{j-1} = 0$, $u_j = E - 1$ et $u_{j+1} = 0$, on divise $E^2 - E$ par $2E - 1$, le quotient est $E/2 - 1$, et on l'estime à $E - 1$. Dans le cas $E = 2^{32}$, cela fait un grand nombre de corrections. On va donc supposer que v_0 est assez grand, disons $v_0 \geq E/e$, et montrer que la correction est au plus e.

Dans le cas simple où E est une puissance de e, il suffit de multiplier u et v par une certaine puissance de e pour obtenir ce résultat. Dans le cas général, on peut toujours multiplier par un facteur f, quotient de E^2 par $v_0E + v_1 + 1$. Nous laisserons au lecteur le soin de montrer que ceci donne souvent $e = 2$, et $e = 3$ dans les autres cas (il faut supposer E assez grand, disons $E > 20$, pour avoir le cas générique. Les cas E trop petits ne nous intéressent pas ici). L'hypothèse de normalisation faite ici est $e = 2$.

Pour montrer $q - e \leq q_j$ on procède comme suit : si on ajoute eEv_0 aux deux membres de (3), le membre de gauche devient $\geq E^2$. Le membre de droite est $< v_0E(q_j - q + e) + q_j(v_1 + \beta) + v_0E + v_1 + \beta$. Si $q_j < q - e$, on aurait $q_j - q + e + 1 \leq 0$, le membre de droite serait $< (q_j + 1)(v_1 + \beta)$. Or $q_j < E$ car $q \leq E$, $v_1 < E$, le membre de droite serait $< E^2$, absurde.

Supposons maintenant $q = q_j$. Alors (3) se simplifie en

$$rE + u_{j+1} - v_1q_j + \alpha = q_j\beta + r_0E + r_1 + \gamma. \tag{4}$$

Le membre de droite est ≥ 0, d'où on déduit $rE + u_{j+1} \geq v_1q_j$. A contrario, si $rE + u_{j+1} < v_1q$ c'est que q est trop grand. Posons alors $r' = r + v_0$, $q' = q - 1$. On a la relation :

$$r'E + u_{j+1} + \alpha = v_0E(q_j - q') + q_j(v_1 + \beta) + r_0E + r_1 + \gamma. \tag{5}$$

C'est la même relation que (3).

Itérons cette procédure de diminution de q jusqu'à avoir

$$rE + u_{j+1} \geq v_1q. \tag{6}$$

Supposons que $q_j = q - k$. Par hypothèse $k \geq 0$. Alors

$$rE + u_{j+1} - v_1q = r_0E + r_1 - k(v_0E + v_1) + q_j\beta + \gamma - \alpha \geq 0.$$

L'hypothèse H_2 donne

$$(q_j + 1)\beta > \alpha + (k - 1)(v_0E + v_1).$$

On a $q_j + 1 \leq E$, donc $(q_j + 1)\beta - \alpha < E$, et comme $v_0 > 0$, ceci donne $k - 1 \leq 0$, donc $k \leq 1$. Notons que si $\beta = 0$, on obtient $k - 1 < 0$, d'où $k = 0$, i.e., $q = q_j$.

Cette relation est bien entendu utilisée si $\beta = 0$. Dans ce cas, on écrit $rE + u_{j+1} = r'_0E + r'_1 + CE^2$, avec $C = 0$, ou $C = -1$. La contrainte (6) est $C = 0$. Si tel n'est pas la cas, on ajoute $v_0E + v_1$ à $r'_0E + r'_1$ jusqu'à ce qu'il y ait dépassement de capacité, donc une retenue de 1, qui additionnée à C donne 0. Dans `two_div` et `divide_two_digits`, c'est ainsi que l'on procède. La subtilité est que la retenue de la soustraction est $1+C$. On a donc une boucle interne « tant que la retenue est nulle, incrémenter $r'_0E + r'_1$ ». Dans la procédure `ex_quo_i`, v_1q est noté A, $rE + u_{j+1}$ est noté B, $v_0E + v_1$ est noté c, $r'_0E + r'_1$ est noté r, et on ajoute c à r tant que $r \geq c$, si $A > B$. Rappelons que dans ce cas, $E = 2^{16}$, et les diverses quantités A, B, etc., tiennent sur 32bits.

Dans le cas général, on procède comme suit : si $u_{j-1} = v_0$, on pose $q = E$, $r = u_j$. Comme on veut $q < E$, on met 0 dans q et on positionne une variable *spec* à vrai (elle est fausse sinon). On déclare de plus que la relation (6) est fausse dans ce cas. Sinon, on calcule q et r par division, on teste (6). Si la relation est fausse, on décrémente q, et on incrémente r de v_0. S'il y a overflow, la relation (6) est automatiquement vérifiée. De plus, le membre de gauche de (5) est alors $\geq E^2$. Le membre de droite est $< v_0E(q_j - q' + 1) + (q_j + 1)(v_1 + \beta)$, en utilisant H_2. Or $(q_j + 1)(v_1 + \beta) < E^2$, il faut donc $q_j - q' + 1 > 0$, donc $q' \leq q_j$. Comme on avait $q' \leq q_j$, on a égalité. Le fait qu'il y ait overflow est gardé dans la variable *OVF*.

Supposons $q = 0$. Comme on sait $0 \leq q_j \leq q$, c'est que q est exact. De plus il n'y a pas de soustraction à faire. Dans le cas contraire, faisons la soustraction.

$$(u_{j-1} \dots u_{j+m}) - q(v_0v_1 \dots v_m) = (r'_0r'_1 \dots r'_m).$$

Le calcul se fait de la façon suivante : soit C_k la retenue courante, initialisée par $C_m = 0$. On écrit

$$r'_k + qv_k + C_k = EC_{k-1} + u_{j+k}, \tag{7}$$

ou encore

$$C_k + qv_k + (E - 1 - u_{j+k}) = EC_{k-1} + (E - 1 - r'_k). \tag{8}$$

La relation (8) est utilisée dans l'algorithme à l'étape 2.4.2. On calcule le membre de gauche ce qui donne le membre de droite sous la forme $EC_{k-1} + x$. Si $y = E - 1 - x$, alors $E - 1 - y = x$, donc y est r'_k. La relation (7) est utilisée pour comprendre ce que l'on fait. Posons $y = C_0$, multiplions (7) par E^{m-k} et sommons. Il vient

$$(r'_1 \dots r'_m) + q(v_1 \dots v_m) = (u_{j+1} \dots u_{j+m}) + (y0 \dots 0).$$

En comparant avec (3) on obtient

$$Er + r'_1 + \gamma' = (q_j - q)(Ev_0 + v_1 + \beta) + (r_0 + y)E + r_1 + \gamma. \tag{9}$$

Supposons $q = q_j$. Alors $r'_i = r_i$ pour $i > 0$, et $\gamma = \gamma'$. Il reste $Er = (r_0 + y)E$, donc $r = r_0 + y$, $r_0 = r - y$, et $r \geq y$.

Dans le cas contraire, on sait que $q_j = q - 1$, donc

$$Er + r'_1 + \gamma' + Ev_0 + v_1 + \beta = (r_0 + y)E + r_1 + \gamma. \tag{10}$$

ou

$$E(r - y) + r'_1 + \gamma' = r + 0E + r_1 + \gamma - (Ev_0 + v_1 + \beta)$$

Par H_2 le membre de droite est < 0 donc $r < y$.

On en déduit le critère suivant : $q = q_j$ si et seulement si $r \geq y$. Calculons alors $1 + (E - 1 - y) + r = Ea + b$ (étape 2.4.5). Si $r \geq y$, on a $a = 1$, $b = r - y$, d'où $r_0 = b$. Donc si $a = 1$, c'est que $q = q_j$, $r'_i = r_i$ pour $i > 0$ et $r_0 = b$. L'étape de division est donc terminée. Remarquons que si $OVF = 1$, la vraie quantité r est en fait $E + B$, et l'on calcule $1 + (E - 1 - y) + B = r - y = r_0$. Comme on l'a déjà noté, dans ce cas q est correct.

Supposons maintenant $a = 0$, et posons $r'_0 = E - y + r$. Alors (10) devient

$$Er'_0 + r'_1 + \gamma' + Ev_0 + v_1 + \beta = E^2 + Er_0 + r_1 + \gamma.$$

En d'autres termes, les r_i se calculent en additionnant v et r'. La retenue finale est à ignorer (à cause du terme E^2).

Algorithme 9. (Division interne) *On utilise comme variables et paramètres* V_0, V_1, U_j, W, OVF, r, q *et* l, *des chiffres*, N, j, i_q, M, L *et* k, *des indices*, first, spec *et* ok *des booléens. Les deux données* u *et* v *sont des tableaux de chiffres.* [La sémantique de l'algorithme est donnée au début de la section.]

Procédure iquomod8. *Données* u *et* v.

1. *Poser* $N = t_B - i_B$, $j = i_A$, $r = 0$, $i_q = i_C$, $m = t_B$, $V_0 = v[i_B]$, $V_1 = v[i_B + 1]$, $M = t_A - N$, first = *vrai*, spec = *faux*.

2. *Répéter* $M - i_A + 1$ *fois*

2.1. *Poser* $U_j = u_j$ *et* $W = u_{j+1}$. *Poser* ok = *faux*, OVF = 0.

2.2.a. *Si* $r = V_0$, *poser* $q = 0$, $r = U_j$, spec = *vrai*.

2.2.b. *Dans le cas contraire, poser* $\mathtt{C} = r$, $q = \mathtt{ex/}(U_j, V_0)$ *et* $r = \mathtt{C}$.

2.3. *Tant que* ok *est faux* [Première correction du quotient.]

2.3.1. *Si* spec *est faux, alors : poser* $\mathtt{C} = 0$, *soit* $l = \mathtt{ex*}(q, V_1, 0)$. *Positionner* ok *à vrai si* $\mathtt{C} < r$, *à faux si* $\mathtt{C} > r$, *et si* $\mathtt{C} = r$, *à vrai si* $l \leq W$.

2.3.2. *Si* spec *est vrai ou si* ok *est faux, décrémenter* q, *positionner* spec *à faux,* C *à 0,* r *à* $\mathtt{ex+}(r, V_0)$ *et* OVF *à* C. *Si* C *est non nul, mettre* ok *à vrai.*

2.4. *Si* q *est non nul* [soustraction]

2.4.1. *Poser* $L = m$, $k = N + j$, $\mathtt{C} = 0$.

2.4.2. *Tant que* $L > i_B$, *remplacer* u_k *par* $\mathtt{ex-}(\mathtt{ex*}(v_L, q, \mathtt{ex-}(u_k)))$, *décrémenter* L *et* k.

2.4.3. *Soit* $l = \mathtt{ex-}(\mathtt{C})$.

2.4.5. *Poser* $\mathtt{C} = 1$. *Remplacer* u_j *par* $\mathtt{ex+}(r, l)$.

2.4.6. *Si* $\mathtt{C} \neq 0$ *et* OVF *est 0* [seconde correction du quotient]

Décrémenter q.

Poser $L = m$, $k = N + j$, $\mathtt{C} = 0$.

Tant que $L \geq i_B$, *remplacer* u_k *par* $\mathtt{ex+}(u_k, v_L)$, *décrémenter* k *et* L.

2.5. *Poser* $r = u_j$, *puis* $u_j = 0$.

2.6. *Si* first = *vrai et* $q = 0$, *ne rien faire. Sinon, poser* first = *faux, positionner* q *dans* u *à la position* i_q, *et incrémenter* i_q.

2.7. *Incrémenter* j.

3. *Poser* $u_{j-1} = r$.

4. *Poser* $t_B = t_A$, $t_A = i_q - 1$, *et* $i_B =$ `afind0_beg`(u, M, t_B).

Procédure two_div. *Paramètres* u *et* v [cas où v a deux chiffres].

1. *Poser* $i = i_A$, $j = i_C$, first = vrai. *Soient* V_0 *et* V_1 *les deux chiffres de* v, `C` $= 0$ *et* $U_1 = u_i$.
2. *Répéter* $t_A - i_A$ *fois*
  2.1. *Incrémenter* i, *poser* $U_2 = u_i$.
  2.2.a. *Si* `C` $= V_0$, *poser* $q = 0$, `C` $= 1$, $r_1 = U_2$ *puis* $r_0 =$ `ex--`(U_1, V_1).
  2.2.b. *Sinon poser* $q =$ `ex/`(U_1, V_0), $r =$ `C`, `C` $= 0$, $a_1 =$ `ex*`$(q, V_1, 0)$, $b_1 =$ `C`, `C` $= 1$, $r_1 =$ `ex--`(U_2, a_1) *et* $r_0 =$ `ex--`(r, b_1).
  2.4.6. *Tant que* `C` $= 0$, *décrémenter* q, *poser* $r_1 =$ `ex+`(r_1, V_1) *et* $r_0 =$ `ex+`(r_0, V_0).
  2.6. *Si* first = vrai *et* $q = 0$, *ne rien faire. Sinon, poser first* = faux, *positionner* q *dans* u *à la position* j, *et incrémenter* j.
  2.8. *Poser* $U_1 = r_1$ *et* `C` $= r_0$.
3. *Mettre* r_0 *dans* u *en position* $i - 1$, *et* r_1 *en position* i.
4. *Poser* $t_B = t_A$, $t_A = j - 1$, *et* $i_B = i - 1$ *si* r_0 *est non nul,* i *si* r_0 *est nul mais pas* r_1, *et* $i + 1$ *si les deux sont nuls.*

Procédure divide_two_digits. *Arguments* x, x', y, y'. [Diviser $xE + x'$ par $yE + y'$. Rend le quotient. S'il est $\geq E$, met la variable globale `OVF` à vrai. Ceci ne peut se produire que si $y \neq 0$.]

1. *Si* $y = 0$
  1.1. *Si* $x \geq y'$, *mettre* `OVF` *à vrai, rendre 0.*
  1.2. *Poser* `C` $= x$.
  1.3. *Rendre* `ex/`(x', y').
2. *Poser* $p =$ `anormalise`(y), `C` $= 0$.
3. *Si* $p \neq 0$ [multiplication, division par décalage]
  3.1. *Poser* $t = y'$, $y' = y'2^p$, $y = y2^p + t/2^{32-p}$.
  3.2. *Poser* $t = x'$, $x' = x'2^p$, `C` $= x/2^{32-p}$, $x = x2^p + t/2^{32-p}$.
4. *Soit* $q =$ `ex/`(x, y), $r =$ `C`.
5. *Poser* `C` $= 0$, $a_1 =$ `ex*`$(q, y', 0)$, $b_1 =$ `C`.
6. *Poser* `C` $= 1$, $r_1 =$ `ex--`(x', a_1), $r_0 =$ `ex--`(r, b_1).
7. *Tant que* `C` *est nul*
  7.1. *Décrémenter* q.
  7.2. *Poser* $r_1 =$ `ex+`(r_1, y'), $r_0 =$ `ex+`(r_0, y).
8. *Rendre* q.

Procédure divide_spec. *Arguments* y, y'. [Diviser E^2 par $yE+y'$. Il y a overflow si et seulement si $yE + y' \leq E$.]

1. *Si* $y = 0$ *mettre* `OVF` *à vrai, rendre 0. Idem si* $y = 1$ *et* $y' = 0$.
2. *Poser* $p =$ `anormalise`(y), `C` $= 1$, $x = x' = 0$.
3. *Si* $p \neq 0$ [multiplication, division par décalage]
  3.1. *Poser* $t = y'$, $y' = y'2^p$, $y = y2^p + t/2^{32-p}$.
  3.2. *Poser* `C` $= 2^p$.
4. *La suite est identique à* `divide_two_digits`.

Division générale

Maintenant que nous avons fait le plus gros du travail, nous pouvons expliquer l'algorithme de division. Cet algorithme rend le quotient et positionne le reste dans la variable globale `mod` (en fait la *cval* de la variable Lisp `#:ex:mod`).

La seule hypothèse que nous allons faire en divisant x par y est que y est non nul. On va donc commencer par comparer x et y, car dans le cas $x = y$, le quotient est 1 et le reste est 0, et dans le cas $x < y$, le quotient est 0 et le reste est x. Si y est un chiffre, on utilise la procédure `nquoc` décrite précédemment.

Supposons donc que x et y sont des vecteurs de chiffres, entre 0 et t_1 pour x, entre 0 et t_2 pour y. Supposons de plus que le chiffre en position s de y est non nul, et qu'il n'y a que des 0 après ; en d'autres termes y est YE^m. Décomposons $x = XE^m + Z$. Écrivons alors $X = QY + R$ par division euclidienne. Alors le quotient de x par y est Q et le reste est $RE^m + Z$.

Supposons d'abord $s = 0$, un seul chiffre dans Y, et $t_1 = t_2$, un seul chiffre à considérer dans X. Le quotient et le reste Q et R sont donc triviaux. On est cependant astucieux en ce qui concerne la gestion de la mémoire pour le calcul du reste r.

Supposons maintenant $s = 0$ et $t_1 \neq t_2$. La division va se faire par `iquoc`. On calcule i_B et t_B, les indices à considérer pour `iquoc`. Si `iquoc` rend un reste non nul, le vrai reste est dans le vecteur X entre les indices $t_B + 1$ et t_1. On ne connait pas le nombre exact de chiffres du reste, il faut le calculer. Par contre, si R est non nul, le nombre de chiffres du reste est connu. On va extraire le quotient du vecteur X, y positionner R, puis extraire le reste (procédure `handle_q0`).

Dans le cas général on a $s > 0$. On va appeler l'algorithme général de division sur X et Y. Notons que cet algorithme suppose que Y est normalisé. Soit N le facteur de normalisation. On va alors écrire $NX = Q'(NY) + R'$. En d'autres termes, $Q = Q'$ et $R = R'/N$. Comme multiplier X par N peut augmenter sa taille et que l'algorithme de division demande un chiffre de libre, on copie X (en fait x) dans un vecteur plus long de deux cases (donc de taille $t_1 + 3$). Comme la multiplication va écraser Y, on copie Y (en fait y) si $N \neq 1$.

On positionne alors les quantités suivantes : i_A et t_A les indices de X dans la copie de x, i_B et t_B les indices de Y dans y (ou sa copie), et t_C le dernier indice de x dans sa copie.

On va alors appeler `iquomod4`. Cette procédure est coupée en deux pour des raisons purement techniques. Elle va positionner Q et R dans le vecteur X, le quotient est entre les indices i_A et t_A, et le reste entre les indices i_B et t_B. Elle positionne t_B à l'ancien t_A. En ce qui nous concerne, X est x, et juste derrière se trouve Z. En d'autres termes, le reste r est le contenu de x entre les indices i_B et t_C.

La procédure commence par multiplier X et Y par N. Bien entendu, dans le cas $N = 1$, on ne fait rien. La multiplication se fait par décalage de bits. La vraie division se fait par `iquomod8` ou `two_div` suivant que Y a plus de deux chiffres ou non. Finalement, si N n'est pas 1, on divise le reste R' par N par décalage.

Algorithme 10. (Division des entiers) *On utilise comme variables locales et paramètres x et y, des entiers internes, X et Y des vecteurs de chiffres, q un chiffre, h, h' des pointeurs de tas, et t_1, t_2, N, s, i et j des indices.*

Procédure nquomod, fonction principale. *Paramètres x et y.*

1. *Si $x = y$ poser* `mod` $= 0$, *rendre* 1.

2. *Si* $x < y$ *poser* `mod` $= x$, *rendre* 0.

3. *Si* y *est un chiffre, calculer* $q =$ `nquoc`$(x,$`Int_val`$(y))$. *Poser* `mod` $=$ `make_int(C)` *et rendre* q.

4. [Ceci est la division en général] *Dans les autres cas, la procédure est comme suit.*

4.1. *Poser* $t_1 = t(x) - 1$, $t_2 = t(y) - 1$.

4.2. *Soit* $s =$ `afind0_end`$($`UHEAP`$(y), 0, t_2)$.

4.2.1. *Si* $s = 0$ *et* $t_1 = t_2$, *rendre* `iquomod2`(x, y, t_1).

4.2.2. *Si* $s = 0$ *et* $t_1 \neq t_2$, *copier* x, *poser* $t_B = t_1 - t_2$, $i_B = 0$, *et* $t_C = t_1$. *Appeler* `iquoc` *sur* `UHEAP`(x) *et* y_0, *rendre* `handle_q0`(x).

4.2.3. *Si* $s > 0$, *soit* $N =$ `anormalise`(y_0). *Si* N *est non nul copier* y.

4.2.3. *Poser* $i_B = 0$, $t_B = s$, $t_A = 2 + t_1 - (t_2 - s)$, $i_C = 0$, $i_A = 2$, $t_C = t_1 + 2$.

4.2.3. *Copier* x *dans un vecteur de taille* $t_1 + 3$, *cadré à droite.*

4.2.3. *Appeler* `iquomod4` *sur* `UHEAP`(x), `UHEAP`(y) *et* N.

4.2.3. *Mettre* `nunderflow4`(x, i_B, t_C) *dans* `mod`.

4.2.3. *Rendre* `nunderflow3`(x, i_A, t_A).

Procédure iquomod2. *Paramètres* X, Y *et* s [on suppose $X = X_0 E^s + r$ et $Y = Y_0 E^s$].

Poser `C` $= 0$, $q =$ `ex/`(X_0, Y_0).

Si `C` $\neq 0$, *copier* X, *mettre* `C` *dans* X_0, *et mettre* X *dans* `mod`.

Sinon, mettre `nunderflow3`$(X, 1, s)$ *dans* `mod`.

Rendre `make_int`(q).

Procédure handle_q0. *Données* X. [Extrait le quotient et le reste du vecteur X.]

Si `C` $\neq 0$, *allouer un vecteur* Y *de taille* $t_C - t_B + 1$. *En position 0, y mettre* C, *y copier* X *via* `bltvector`$(Y, 1, X, t_B + 1)$. *Mettre* Y *dans* `mod`.

Sinon, mettre `nunderflow3`$(X, t_B + 1, t_C)$ *dans* `mod`.

Rendre `nunderflow3`(X, i_B, t_B).

Procédure iquomod4. *Paramètres* h, h' *et* N. [Multiplie X et Y par 2^N avant de diviser. La multiplication de X peut faire un overflow.]

Si N *est non nul*

Appeler `shift_left`(h', i_B, t_B, c, N).

Appeler `shift_left`(h, i_A, t_A, c, N).

Si c *est non nul, décrémenter* i_A, *mettre* c *dans* h *à la position* i_A.

Appeler `iquomod8t` *sur* h, h' *et* N.

Procédure iquomod8t. *Paramètres* h, h' *et* N. [Divise X par Y. Le reste est ensuite divisé par 2^N. Il peut y avoir underflow.]

Appeler `two_div` *ou* `iquomod8` *suivant que* h' *a ou non 2 chiffres* [le nombre de chiffres est $t_B - i_B + 1$].

Poser $i_A = i_C$.

Si $i_B \leq t_B$ *et* N *non nul, appeler* `shift_right` *sur* h, i_B, t_B, c *et* N. *Si* $h[i_B]$ *est nul, incrémenter* i_B. [Si $I_B > t_B$, le reste est nul, il n'y a rien à faire.]

2.7 Impression

Le premier algorithme que nous allons donner imprime un entier en base b. Cette base peut n'avoir aucun rapport avec la constante $E = e^p$. On suppose $2 \leq b \leq 36$. Ceci est dû au fait qu'on suppose qu'un chiffre en base b est l'un des 10 chiffres entre 0 et 9, ou l'une des 26 lettres de l'alphabet. On suppose savoir imprimer un chiffre $< b$ (fonction `prin_cn` de Lisp). On suppose également que l'imprimeur Lisp sait imprimer des petits entiers. La procédure que nous allons décrire ici n'imprime donc que des grands entiers. Elle ne teste pas si le nombre tient sur la ligne courante: les caractères sont imprimés les uns après les autres, si la ligne courante est pleine, un changement de ligne est généré automatiquement. La stratégie utilisée par SISYPHE pour imprimer des grands entiers est simplement d'imprimer ces nombres dans le tampon d'impression (nettoyé au préalable) puis d'extraire le résultat du tampon, et d'imprimer ceci comme une chaîne de caractères (ou suite de chaînes).

Tout le problème est donc de calculer les chiffres de notre nombre de façon optimale, en évitant au maximum les divisions. Pour des raisons d'efficacité, le code de la division est une copie du code de `iquoc`.

On suppose que dans deux tables `fbs` et `fbn` se trouvent à la position i pour chaque base i deux entiers j et x où $x = i^j$, et j est la plus grande puissance telle que $x \leq E$. Ces tables sont calculées lors du lancement du programme. On va donc essayer d'obtenir j chiffres d'un coup, par division. Comme le reste du nombre par la base fournit le dernier chiffre à imprimer, il faut ranger ces nombres dans une table. On vérifie si $e = 2$ et $p = 32$, qu'une table de taille $3/2$ fois la taille du nombre est assez grande. Enfin, dans le cas où $x = E$, on met 0 à la place de x dans la table et on ne fait pas de division du tout. On utilise une table `buf16` de taille q telle que $2^{q+1} > E$ (pour toute base, tout nombre $< E$ a au plus q chiffres dans cette base). Ce vecteur est de taille 32 si $E = 2^{32}$. En base 10, on génère les chiffres par paquets de 32, 64, 128, ou même plus, on divise chaque paquet en deux, tant que les paquets occupent plusieurs mots machine.

Algorithme 11. (Impression des entiers) *On utilise comme variables, des chiffres x, x_1, x_2, x_3, x_4, c, des tableaux de chiffres X, B, des indices i, j, k, s, I, des entiers internes y, q, r, et un vecteur de pointeurs C. On utilise deux variables globales* `fbs`, `fbn` *initialisées au lancement, un tableau de taille 32* `buf16`, *et la base de sortie courante* `obase`, *notée b.*

Procédure nprin0, procédure principale. *Paramètre n* [n est un vrai nombre].

Si n est un grand entier, le remplacer par `rz_num`(n).

Si n est un chiffre, appeler `aprin_no_just`(`Int_val`(n)).

Si la base est 10, appeler `abase10prin`(n).

Dans les autres cas, soit j (resp. x) le contenu de `fbs` *(resp.* `fbn`*) pour la base courante, et k la taille de n.*

Si la base est 2, 4 ou 16,

> *Appeler* `aprin_no_just` *sur n_0.*
>
> *Pour i de 1 à $k-1$, exécuter* `aprin_just`(n_i, j).

Sinon, soit B le pointeur de tas d'un vecteur de chiffres de taille $3/2k$. Copier n. Appeler `iprin0`(`UHEAP`$(n), k, B, j)$. *Si le résultat est s, appeler* `aprin1`(B, s, x).

Procédure aprintdigit. *Paramètre c, un entier entre 0 et 35.* [ceci suppose que les lettres de l'alphabet ont des code internes consécutifs, par exemple, codes ASCII.]

Si $c < 10$, ajouter à c le code interne de 0. Sinon ajouter à c le code interne de 'A' moins 10.

Considérer c comme un caractère.

L'imprimer via `prin_cn`.

Procédure output_prin_vect. *Paramètre i* [Imprimer les i premiers chiffres de `buf16`].

Tant que $i > 0$ décrémenter i et imprimer le chiffre en position i de `buf16` *via* `aprintdigit`.

Procédure aprin_just. *Paramètres x et j* [$x < E$, imprimer x avec j chiffres].

Pour i entre 0 et $j - 1$ mettre le reste de x par b dans `buf16` *en position i, remplacer x par le quotient* [quotient, reste via division normale 32bits non signée].

Appeler `output_prin_vect` *sur j.*

Procédure aprin_no_just. *Paramètre x* [$x < E$, imprimer x].

Si $x = 0$, imprimer le chiffre 0.

Sinon, soit $i = 0$, tant que $x \neq 0$ mettre le reste de la division de x par b dans `buf16` *en position i, remplacer x par le quotient, et incrémenter i.*

Appeler `output_prin_vect` *sur i.*

Procédure aprin1. *Paramètres X, i, j.*

Imprimer X_i via `aprin_no_just`.

Imprimer X_k $(k = i - 1, \ldots, 0)$ via `aprin_just`(X_k, j).

Procédure iprin0. *Paramètres X, s, B et x* [s est la taille du tableau de chiffres X, B le tampon, x une puissance de la base de sortie].

Poser $I = 0$, $i = 0$, $k = 0$.

Poser $p =$ `anormalise`(x). *Multiplier x par 2^p par décalage. Écrire $x = x' + x''2^{16}$ en utilisant* `HIBITS` *et* `LOBITS`.

Tant que $I < s$

Si p est non nul, calculer `shift_left`$(X, I, s-1, c, p)$ *et mettre c dans* `C`. *Sinon poser* `C` $= 0$.

Poser $i = I$, $X_i =$ `ex_quo_i`$(\texttt{C}, X_i, x, x', x'')$, *et incrémenter i.*

Si $X_{i-1} = 0$, poser $I = i$.

Tant que $i < s$ remplacer X_i par `ex_quo_i`$(\texttt{C}, X_i, x, x', x'')$ *et incrémenter i.*

Mettre `C`$/2^p$ *dans B en position k, incrémenter k.*

Rendre $k - 1$.

Procédure aprin10n. *Argument x et k* [Imprimer x en base 10 avec k chiffres].

Poser $t = x$. Tant que $t \neq 0$, diviser t par 10, décrémenter k.

Imprimer k chiffres 0.

Si x est non nul, imprimer x via `aprin_no_just`.

Procédure aprin10_8. *Paramètres x et s.*

Si s est vrai, appeler `aprin10n`$(x, 8)$.

Sinon appeler `aprin_no_just`(x).

Procédure aprin10_32. *Paramètres y et s.* [Imprimer y sur 32 chiffres en base 10 si s est vrai, au plus 32 chiffres sinon.]

1. *Si y est un chiffre et s vrai, appeler* `aprin10n`(`Int_val`$(y), 32)$.
2. *Si y chiffre, s faux, appeler* `aprin_no_just`(`Int_val`(y)).
3. *Dans les autres cas, diviser y trois fois par 10^8. Soient x_4, x_3, x_2 les* `Int_val` *des trois restes, x_1 le dernier quotient.* $[y = x_1 10^{24} + x_2 10^{16} + x_3 10^8 + x_4]$
4. *Imprimer x_1, x_2, x_3 et x_4. Si s est vrai, on utilise* `aprin10_8`(x_i, s), *sinon, le premier non nul est imprimé avec* `aprin10_8`$(x_i, 0)$, *les suivants avec* `aprin10_8`$(x_i, 1)$.

Procédure aprin10_64. *Paramètres y et s.* [Imprimer y sur 64 chiffres en base 10 si s est vrai, au plus 64 chiffres sinon.]

1. *Si y est un chiffre et s vrai, appeler* `aprin10n`(`Int_val`$(y), 64)$.
2. *Si y chiffre, s faux, appeler* `aprin_no_just`(`Int_val`(y)).
3. *Dans les autres cas, calculer le quotient q et le reste r de la division de y par 10^{32}.*
4. *Si $q = 0$ et $s = 0$, imprimer r via* `aprin10_32`$(r, 0)$.
5. *Sinon imprimer q via* `aprin10_32`(q, s), *et r via* `aprin10_32`$(r, 1)$.

Procédure abase10prin. *Paramètre y.*

1. *Si y est un chiffre poser $k = 1$, sinon $k = 1 + t(y)/6$.*
2. *Mettre dans C un vecteur de taille k* [si $k = 1$, ce vecteur est alloué une fois pour toutes.]
3. *Tant que y est non nul, diviser y par 10^{64}, mettre les reste dans C, remplacer y par le quotient.* [la division est via `nquomod`, l'entier interne associé à 10^{64} est calculé au lancement du programme]
4. *Imprimer les éléments du vecteur C, via* `aprin10_64`(C_i, s), *en commençant par le dernier avec $s = 0$, les autres avec $s = 1$.*

Autre algorithme d'impression. Cet algorithme imprime les p premiers chiffres d'une fraction en base b. Si le nombre est périodique, la période est entre accolades. Par exemple, 19/6 sera imprimé comme `3.1{6}`. Dans le cas $p < 0$, on imprimera exactement $-p - 1$ chiffres, sans indiquer la période. Les chiffres à gauche du point s'obtiennent en imprimant la partie entière du quotient n/d, donc q si $n = dq + r$.

Il s'agit alors d'imprimer la partie fractionnaire. Supposons que n et d soient premiers entre eux. C'est cette condition qui va nous permettre de trouver la période. Soit x le premier chiffre imprimé, et n'/d' ce qui reste à imprimer après le premier chiffre. On a donc la relation

$$\frac{n}{d} = \frac{x}{b} + \frac{n'}{bd'}.$$

Cette relation s'écrit aussi $nb = dx + dn'/d'$. On veut $0 \leq x < b$ et $0 \leq dn'/d' < d$. En d'autres termes, x est le quotient de la division de nb par d; si $bn = xd + r$, on a $d'r = dn'$. Comme on veut que d' soit premier avec n', c'est que d' divise d, donc $d = \lambda d'$. On a donc $r = \lambda r'$. Or $bn = \lambda(d'x + r')$, donc λ divise b, donc μ le pgcd de b et d. Réciproquement, si $\lambda = \mu$, et si on calcule $nb/\mu = x(d/\mu) + n'$, alors n' et $d' = d/\mu$ sont premiers entre eux. Comme la base b est au plus 36, on préfère calculer le pgcd de b et d, plutôt que de calculer le pgcd de d et r.

Notons qu'à l'itération suivante, on calculera $\mathrm{pgcd}(b, d/\mu)$. Si p est un nombre premier divisant ce pgcd, et premier à μ, il divisera d et b, donc μ, absurde. On en déduit que ce pgcd sera trivial à partir du moment où $\mu = 1$. En fait, on divise d par toutes les puissances des facteurs premiers de b jusqu'à obtenir à un certain moment $\mathrm{pgcd}(b, d) = 1$.

Ce phénomène se produit nécessairement (sauf si n s'annule, auquel cas le nombre est complètement imprimé). Supposons que N est la valeur de n au moment où le pgcd μ devient 1. Ce que l'on fait ensuite est de calculer $bn = dx + r$, et de remplacer n par r. Au bout de k itérations n est remplacé par $b^k N$ modulo d. Comme b et N sont premiers à d, cette quantité n'est jamais nulle. De plus, b est un élément du groupe inversible modulo d, lequel groupe est de cardinal $\phi(d)$. Donc $b^{\phi(d)} = 1$ modulo d. En particulier, la suite des n (donc la suite des x) est périodique, de période $\phi(d)$. La plus petite période peut bien entendu être un diviseur strict de $\phi(d)$, par exemple 7/9 est `0.{7}` en base 10, `0.{61}` en base 8, `0.{530}` en base 7, et `0.{342102}` en base 5, et $\phi(9) = 6$.

Si la période commençait avant que μ ne devienne 1, on aurait les relations suivantes. Soit x le dernier chiffre de la période. On a $bn' = dx + r'$, et $r' = N$. On fait l'hypothèse que $n''b/\mu = x(d''/\mu) + r''$, et que $d''/\mu = d$ et que $r'' = N$. On en déduit $n'\mu = n''$, donc μ divise n''. Or μ divise d'', μ est non trivial, et n'' et d'' sont premiers entre eux, absurde. On sait donc localiser le début de la période. Ceci fournit l'algorithme suivant.

Algorithme 12. (Écriture décimale) *Données x, et p. Les variables locales n, d, q, r, et N sont des entiers internes, p et s des indices, et x un nombre* [On imprime le quotient n/d avec au plus p chiffres après la virgule].

1. *Écrire $x = \pm n/d$, si x est un entier ou une fraction rationnelle, erreur dans le cas contraire. Ici n et d sont des entiers internes.*

2. *Imprimer le signe si $x < 0$.*

3. *Écrire $n = qd + r$ via* `nquomod`*, imprimer q via* `nprin0`*, et poser $n = r$.*

4. *Fin de la procédure si $n = 0$. Sinon imprimer un point.*

5. *Si $p < 0$, remplacer p par $-p - 1$, mettre s à 1 sinon à 0.*

6. *Poser $q = \mathrm{pgcd}(b, d)$, $B = b/q$, et $d = d/q$.*

7. *Faire tant que $q \neq 1$*

7.1. *Écrire $nB = dq + r$, imprimer q via* `aprintdigit`*, poser $n = r$, décrémenter p.*

7.2. *Fin de la procédure si $n = 0$.*

7.3. *Si $p = 0$: imprimer "..." si $s = 0$ et fin.*

7.4. *Refaire* **6.**

8. *Si $s = 0$: poser $N = n$, imprimer une accolade ouvrante.*

9. *Tant que* $p > 0$ *faire*

9.1. *Refaire* **7.1.**

9.2. *Si* $s = 1$ *et* $p = 0$, *fin.*

9.3. *Si* $s = 0$ *et* $p = 0$, *imprimer trois petits points.*

10. *Si* $s = 0$ *imprimer une accolade fermante.*

2.8 Pgcd

On propose l'algorithme suivant de calcul de pgcd.

Algorithme 13. (Pgcd sur les entiers positifs) *On utilise comme variables locales et paramètres* a *et* b, *des nombres internes,* x, y, N, *des chiffres,* u, v *des vecteurs de chiffres,* U, V *des tableaux de chiffres, et* i, j *des indices.*

On utilise une variable globale Z *qui est un vecteur de chiffres, et* Z' *son pointeur de tas.*

Procédure npgcd, fonction principale. *Paramètres* a *et* b.

Comparer a *et* b *via* `ncmp`.

Si $a = b$, *rendre* a.

Si $a < b$ *appeler* `ipgcd3` *sur* b *et* a.

Sinon appeler `ipgcd3` *sur* a *et* b.

Procédure ipgcd3. *Données* a *et* b [On suppose $a > b$].

Si a *est un chiffre rendre* `make_int(ipgcd0(Int_val(`a`),Int_val(`b`)))`.

Si b *est un chiffre, poser* $i_B = 0$, $t_B = t(a) - 1$, *copier* a, *et rendre* `make_int(ipgcd1(`a`,Int_val(`b`)))`.

Sinon, soit $s = t(a) + 2$. *Allouer un vecteur* Z *de taille* s. *Mettre dans* Z' *le pointeur de tas de* Z. *Copier* a *et* b *dans des vecteurs de taille* s, *cadrés à droite.*

Poser $i_A = 2$, $t_A = s - 1$, $t_B = s - 1$, $i_B = s - t(b)$.

Appeler `ipgcd2` *sur* a *et* b.

Nettoyer Z', *rendre le résultat.*

Procédure ipgcd0. *Données* x *et* y.

Si $y = 1$, *rendre* 1.

Remplacer x *par le reste de la division de* x *par* y [division machine].

Si $x = 0$ *rendre* y, *sinon* `ipgcd0(`y,x`)`.

Procédure ipgcd1. *Données* u *et* y.

Appeler `iquoc` *sur* u *et* y.

Si `C` $= 0$, *rendre* y, *sinon* `ipgcd0(`y`,C)`.

Procédure ipgcd2. *Données u et v* [On suppose $u > v$, u entre i_A et t_A, v entre i_B et t_B].

Appeler `gcd_via_bezout`$(\texttt{UHEAP}(u), \texttt{UHEAP}(v), 0)$.

Si $t_B < i_B$, *rendre* `nunderflow3`(v, i_A, t_A).

Si $t_B = i_B$, *poser* $y = u_{i_B}$, $i_B = i_A$, $t_B = t_A$, *et rendre* `make_int`$(\texttt{ipgcd1}(\texttt{UHEAP}(v), y))$.

Sinon rendre `ipgcd2`(v, u).

Procédure gcd_via_div. *Données U et V.* [u entre i_A et t_A, v entre i_B et t_B ; remplace u par le reste de la division de u et v. Après, i_B et t_B sont les indices du reste, i_A et t_A ceux de v. Les arguments sont les pointeurs de tas de u et v].

Poser $i = i_B$ *et* $j = t_B$, $i_C = 0$.

Soit $N =$ `anormalise`$(V[i_B])$.

Si $N \neq 0$, *copier V dans Z'.*

Appeler `iquomod4` *sur U, V (ou Z' si on a copié) et N.*

Poser $i_C = t_A$.

Poser $i_A = i$ *et* $t_A = j$.

Algorithme de Lehmer

La première implémentation de cet algorithme consistait à utiliser `gcd_via_div` au lieu de `gcd_via_bezout`. Ce que nous allons décrire est la nouvelle version (algorithme de Lehmer).

L'idée de base est que très souvent le quotient calculé est petit. On estime ce quotient en prenant les premiers bits, et on fait plusieurs divisions successives sur ces premiers bits. Au bout de quelques opérations, u est remplacé par $Au+Bv$, et v est remplacé par $Cu+Dv$. Ceci revient à faire 4 multiplications d'un grand entier par un chiffre et deux soustractions. L'algorithme de division calcule essentiellement quatre multiplications et une soustraction (une multiplication interne, deux multiplications pour la normalisation et une division, qui équivaut à une multiplication). Bien entendu, lors de la division, la soustraction et la multiplication se font en une passe, mais on peut faire de même dans ce cas. Si on arrive donc à remplacer plusieurs divisions par une opération de ce type on y gagne. Si on arrive à remplacer une division par une opération de ce type on y gagne souvent (le quotient est connu, on peut éviter la phase de normalisation). Bien entendu, s'il faut quand même faire une division on y perd, mais pas trop.

Notons que les divers quotients apparaissant lors du calcul du pgcd sont les quotients du développement en fraction continue du quotient. Dans [11, §3.6] cette technique est utilisée pour compacter les représentations des réels sous forme de fraction continue.

La première chose à faire est de calculer les premiers bits de u et v, en d'autres termes x et y tels que $u = e^k(x + \hat{x})$ et $v = e^k(y + \hat{y})$, où x et y sont des entiers et $0 \leq \hat{x} < 1$, $0 \leq \hat{y} < 1$. On veut $0 < x < E$, en fait on veut x le plus grand possible. Cette méthode n'a aucun intérêt si l'on prend pour x les bits qui se trouvent dans le chiffre principal de u. On veut donc $E/e \leq x < E$. Supposons que u soit dans le vecteur A entre les indices i_A et t_A. Comme on est dans la procédure `ipgcd2`, on sait que $i_A < t_A$. Supposons que les chiffres en position i_A et $i_A + 1$ de u soient α et β. On suppose que les chiffres aux mêmes indices de v sont α' et β'. Dans le cas où $i_A < i_B$, on prend $\alpha' = 0$ (en fait, on met 0 dans v à la position i_A, cela servira dans la suite). Dans le cas où $i_A + 1 < i_B$ on a de plus $\beta' = 0$. Comme nous allons le voir dans la suite, ce cas est sans

intérêt, car il faut faire une division. Quitte à décrémenter i_B, on supposera donc $i_B = i_A$. Notons aussi que $t_B = t_A$, dans la mesure où on a pris la précaution de mettre u et v dans des vecteurs de même taille, cadrés à droite. Une telle condition restera toujours vraie. Cette façon de faire a comme avantage de ne pas être obligé de mettre à jour les indices t_A et t_B.

Pour rendre le code encore plus efficace, on prendra $2p$ (i.e., 64) bits pour x et y. Alors x et y sont représentés par des couples de chiffres. Il faudra dans la suite calculer le quotient et le reste de x par y. Le quotient s'obtient grâce à la fonction `divide_two_digits`, et si q est ce quotient, le reste r est calculé par $r = x - qy$. Dans le cas où $\alpha \geq E/2$, on a $x = \alpha E + \beta$ et $y = \alpha' E + \beta'$. Dans le cas contraire, soient γ et γ' les chiffres de u et v en position $i_A + 2$ (ou 0 si u n'a que deux chiffres). Alors $x = (\alpha E^2 + \beta E + \gamma)/e^q$, $y = (\alpha' E^2 + \beta' E + \gamma')/e^q$ où q est le nombre de bits de α, i.e. $e^{q-1} \leq \alpha < e^q$.

Posons $n = 0$. Appelons x_0 et y_0 les quantités x et y ainsi calculées. On considère des quantités A_n, B_n, C_n et D_n, (initialisées avec 1, 0, 0 et 1 pour $n = 0$) ainsi que $A'_n = (-1)^n A_n$, $B'_n = -(-1)^n B_n$, $C'_n = -(-1)^n C_n$ et $D'_n = (-1)^n D_n$. Pour alléger les notations, on supprimera l'indice n dans la suite. Posons $x' = x + B$, $y' = y + D$, $x'' = x + A$ et $y'' = y + C$. On suppose, pour $n > 0$, et nous le montrerons dans la suite, que ces quantités sont entre 0 et $E^2 - 1$. Dans le cas $n = 0$, on exclus le cas $y' = E^2$; en fait ceci n'est possible que si $x = y$ (car $y \leq x < E^2$). En fait, si les 32 premiers bits de x sont égaux aux 32 premiers bits de y, le premier quotient est 1, le second quotient est très grand, on arrête l'algorithme avec $n = 1$, et on remplace u par $u - v$. Le seul cas problématique est celui où $x = E^2 - 1$, donc $x'' = E^2$. Ce cas est traité à part dans l'algorithme.

Posons $\alpha = uA + vB$ et $\beta = uC + vD$. On a donc $u = \alpha$ et $v = \beta$ pour $n = 0$. On va supposer que, si on faisait effectivement les n divisions sur u et v, les quantités u et v seraient transformées en α et β. Pour deviner les quotients de ces divisions, on suppose que la partie entière de α/e^k est entre x' et x'', celle de β/e^k est entre y' et y''. Comme nous allons le montrer dans la suite, les quantités A', B', C' et D' sont positives ou nulles, donc, si n est pair, on a $x' \leq x \leq x''$ et $y'' \leq y \leq y'$, les inégalités sont dans l'autre sens si n est impair (x et y sont les parties entières de α/e^k et β/e^k).

Écrivons $x' = q'y' + r'$ et $x'' = q''y'' + r''$ par division euclidienne. Si l'une des deux quantités y' ou y'' est nulle, le quotient sera l'infini. On a donc $q' \leq \alpha/\beta < q'' + 1$, donc si $q' = q''$, c'est la partie entière de α/β. Dans le cas où il n'y a pas égalité, on ne connait pas le quotient, et on s'arrête. En particulier, on s'arrête si $y' = 0$, $y'' = 0$ ou si le quotient ne tient pas sur 32bits.

Supposons donc $q = q' = q''$. On pose alors $A_1 = C$, $C_1 = A - qC$, $B_1 = D$, $D_1 = B - qD$, $x_1 = y$ et $y_1 = x - qy$. Les quantités x'_1, etc. s'en déduisent. En particulier, $A'_1 = C'$, $C'_1 = A' + qC'$, $B'_1 = D'$, $D'_1 = B' + qD'$, ce qui montre que ces quatre quantités sont positives ou nulles. L'algorithme s'arrête dans le cas où l'une de ces quantités devient $\geq E$ (c'est pour cela qu'on demande $q < E$).

Notons que $x'_1 = y'$, $x''_1 = y''$, $y'_1 = r'$, et $y''_1 = r''$. Ces relations montrent que ces quantités sont entre 0 et E^2. En fait, E^2 est exclus : on suppose au départ $y' < E^2$ et $y'' < E^2$, et les restes sont plus petits que y', y'', donc plus petits que E^2. Posons maintenant $\alpha_1 = \beta$ et $\beta_1 = \alpha - q\beta$. Des relations $x' \leq \alpha/e^k \leq x''$ et $y'' \leq \beta/e^k \leq y'$ (valides pour n pair) on tire $y'_1 = x' - qv' \leq \beta_1/e^k \leq x'' - qv'' = y''_1$. Le même raisonnement s'applique si n est impair, en renversant le sens des inégalités.

On supprime alors les indices 1, on incrémente n, et on continue. On vient donc de calculer A, B, C, et D tels que $\alpha = uA + vB$ et $\beta = uC + vD$, avec $\beta < \alpha$. Notons que $x = x_0 A + y_0 B$ et $y = x_0 C + y_0 D$. Cette relation s'écrit aussi $x_0 = D'x + B'y$ et $y_0 = C'x + A'y$ (voir l'algorithme

de Bezout général où les quantités A', B', C' et D' s'appellent x, y, u et v). Comme ces deux relations sont entre des nombres positifs ou nuls, on en tire des inégalités importantes: la relation $q' = q''$ impose que A, B, C et D sont plus petits que x et y donc plus petits que $\sqrt{y_0}$. C'est pour cette raison que x_0 et y_0 sont calculés sur 64bits, A, B, C et D sur 32bits.

Il reste maintenant à expliquer comment calculer une quantité de la forme $uA + vB$. En fait, il faut calculer $\pm(uA' - vB')$. Montrons donc comment calculer $uA' - vB'$. On sait que A' et B' sont des chiffres, u et v des vecteurs de chiffres. Supposons d'abord $A' = 1$. Il s'agit de calculer $u - vB'$. On utilise la même procédure utilisée par `iquomod8` pour la soustraction interne. Dans le cas $A' \neq 1$, on calcule $-vB'$ de la même manière, en faisant comme si u était nul. Il s'agit alors de calculer $uA + v'$, ce qui est facile. En fait, on réalise les deux opérations en une seule boucle, en manipulant deux retenues. Finalement, dans le cas $B' = 1$, on calcule $-v$. Ce calcul est trivial: en allant de droite à gauche, on laisse les 0 inchangés. Si v_i est le premier chiffre non nul, on le remplace par $E - v'$. Les autres chiffres sont remplacés par $E - 1 - v_i$.

Finalement, on procède comme suit. Soit n le nombre de divisions faites. Si $n = 0$, et $x = y$, on remplace u par $u - v$. Si $n = 0$ mais $u \neq v$, on divise via `gcd_via_div`. Si $n = 1$, le quotient est D, et on remplace u par $u - Dv$. Sinon on remplace u par $uC + vD$ et v par $uA + vB$. Comme ces opérations sont censées être faites en parallèle, on met le résultat de la première dans Z, le résultat de la seconde dans v, et on copie Z dans u (notons que Z est un vecteur auxiliaire, utilisé pour copier v s'il faut faire une division et normaliser v). Ceci donne l'algorithme suivant.

Algorithme 14. (Pgcd via Lehmer) *Les variables et paramètres utilisés ici sont x, y, x', y', N, A, B, C, D, α, β, α', β', γ, γ' a, b, c, c', d, a_1, b_1, q, q', r_0, r_1, s, w et t des chiffres, u, v et z, des tableaux de chiffres et i, j, k et p, des indices. On utilise de plus la variable globale Z' un tableau de chiffres, et une variable globale booléenne* `OVF`.

Macro lehmer_add. *Paramètres a, b, x, x', A* [Calcule $xE + x' + A$ dans $aE + b$].

1. *Poser* `C` $= 0$.
2. *Poser* $b =$ `ex+`(x', A).
3. *Si* `C` $= 0$, *poser* $a = x$.
4. *Sinon, poser* $a = x + 1$, *et si* $a = 0$, *mettre* `OVF` *à vrai.*

Macro lehmer_sub. *Paramètres c, d, y, y', C* [Calcule $yE + y' - C$ dans $cE + d$].

1. *Poser* `C` $= 1$.
2. *Poser* $d =$ `ex--`(y', C).
3. *Si* `C` $= 1$, *poser* $c = y$.
4. *Sinon, poser* $c = y - 1$, *et si* $y = 0$, *mettre* `OVF` *à vrai.*

Procédure iget_first. *Paramètres u et v.* [Calcul sur 64 bits]

1. *Si* $i_A + 1 < i_B$, *mettre* `OVF` *à vrai. Sinon, si* $i_A < i_B$ *mettre 0 dans v en position i_A.*
2. *Calculer α et β les éléments en position i_A et $i_A + 1$ de u, α' et β' les éléments en position i_A et $i_A + 1$ de v.*
3. *Soit* $w =$ `anormalise`(α), $s = 32 - w$.

4.a. *Si $w = 0$, rendre α, β, α', β'.*

4.b. *Soient γ et γ' les chiffres suivants (0 s'il n'y en a pas).*

4.c. *Poser $\alpha = \alpha 2^w + \beta/2^s$, $\beta = \beta 2^w + \gamma/2^s$, idem pour α' et β'* [multiplication, division par décalage.]

5. *Rendre α, β, α', β'.*

Procédure iget_ABCD. *Paramètres x, x', y et y'* [calcule A'_n, B'_n, C'_n, $D' - N$, n et s, la parité de n].

1. *Poser $s = 1$, $A = 1$, $B = 0$, $C = 0$, $D = 1$, $N = 0$.*

2. *Si $x = y$, poser $N = 1$. Si $x = y$ ou si $y = 0$ et $y' \geq 0$, rendre A, B, C, D, s et N.* [dans le second cas, il y aurait overflow à la division de $xE + x'$ par $yE + y'$.]

3. *Faire ce qui suit, mais si* `OVF` *est vrai, fin de la boucle.*

3.a.1. *Si $s = 1$*

Calculer `lehmer_sub`(c, d, y, y', C).
Calculer `lehmer_add`(a, b, x, x', A).
Si `OVF` *est vrai, $a = b = 0$, poser $q =$* `divide_spec`(c, d).
Sinon poser $q =$ `divide_two_digits`(a, b, c, d).
Calculer `lehmer_add`(c, d, y, y', D).
Calculer `lehmer_sub`(a, b, x, x', B).
Poser $q' =$ `divide_two_digits`(a, b, c, d).

3.a.2. *Si $s = -1$*

Calculer `lehmer_add`(c, d, y, y', C).
Calculer `lehmer_sub`(a, b, x, x', A).
Poser $q =$ `divide_two_digits`(a, b, c, d).
Calculer `lehmer_sub`(c, d, y, y', D).
Calculer `lehmer_add`(a, b, x, x', B).
Poser $q' =$ `divide_two_digits`(a, b, c, d).

3.b. *Si $q \neq q'$, fin de la boucle.*

3.c. *Poser* `C` $= 0$, $c =$ `ex*`(q, C, A), *si* `C` *est non nul, fin. Poser $d =$* `ex*`(q, D, B), *si* `C` *est non nul, fin.*

3.d. *Poser $A = C$, $C = c$, $B = D$, $D = d$.*

3.e. *Poser $a =$* `ex-`(`ex*`$(y', q,$`ex-`$(x'))), b =$ `ex-`(`ex*`$(y, q,$`ex-`$(x)))$,

3.f. *Poser $x' = y'$, $x = y$, $y' = a$, $y = b$.*

3.g. *Remplacer s par $-s$ et incrémenter N.*

4. *Rendre A, B, C, D, s et N.*

Procédure gcd_via_bezout. *Paramètres u et v.*

1. *Calculer x, x', y, y' via* `iget_first`. *Si* `OVF`, *poser $N = 0$, aller en* **3.a.**

2. *Appeler* `iget_ABCD`(x, x', y, y'). *Ceci rend A, B, C, D, s et N.*

3.a. *Si $N = 0$, appeler* `gcd_via_div` *sur u et v.*

3.b. *Si $N = 1$, appeler* `isubtract`(u, v, D).

3.c. *Sinon*

3.c.1. *Si $s = 1$, poser $i_B =$* `isubgcd`(v, D, u, C, Z') *sinon $i_B =$* `isubgcd`(u, C, v, D, Z').

3.c.2. *Si* $s = 1$, *poser* $i_A =$ `isubgcd`(u, A, v, B, v), *sinon* $i_A =$ `isubgcd`(v, B, u, A, v).

3.c.3. *Copier* Z' *dans* u, *via* `ibltvector`$(u, 0, Z', 0, t_A + 1)$

4. *Rendre* A, B, C, D, s, N.

Procédure isubgcd. *Paramètres* u, x, v, y *et* z [Calcule $ux - vy$, le résultat est dans z, les chiffres utiles sont entre i_A et t_A].

1. *Poser* $i = t_A$.

2.a. *Si* $x = 1$, *poser* `C` $= 0$, *tant que* $i \geq i_A$ *remplacer* z_i *par* `ex-`$($`ex*`$(v_i, y,$`ex-`$(u_i)))$ *et décrémenter* i.

2.b. *Si* $y = 1$, *poser* $s = 0$, *tant que* $i \geq i_A$,

Si $s = 1$, *poser* $\alpha =$ `ex-`(v_i).

Si $v_i = 0$ *poser* $\alpha = 0$.

Sinon, poser $s = 1$, $\alpha = -v_i$.

Remplacer z_i *par* `ex*`(u_i, x, α) *et décrémenter* i.

2.c. *Sinon poser* $c = c' = 0$. *Tant que* $i \geq i_A$ *faire*

Poser `C` $= c'$, $\alpha =$ `ex-`$($`ex*`$(v_i, y, -1))$, $c' =$ `C`.

Poser `C` $= c$, *remplacer* z_i *par* `ex*`(u_i, x, α), *poser* $c =$ `C`.

Décrémenter i.

3. *Rendre* `afind0_beg`(z, i_A, t_A).

Procédure isubtract. *Paramètres* u, v *et* x [Calcule $u - vx$, le résultat est dans u, les chiffres utiles sont entre i_A et t_A].

Poser `C` $= 0$ *(*1 *si* $x = 1$*),* $i = t_A$.

Tant que $i \geq i_A$, *remplacer* u_i *par* `ex-`$($`ex*`$(v_i, x,$`ex-`$(u_i)))$ *(ou si* $x = 1$ *par* `ex--`(u_i, v_i)*) et décrémenter* i.

Poser $i_A = i_B$ *et* $i_B =$ `afind0_beg`$(u, i + 1, t_A)$.

Temps de calcul

Nous donnons dans la table qui suit le temps de calcul du pgcd de F_n et F_{n+1} où F_n est le n-ième nombre de Fibonacci. Ce temps est donné en secondes CPU sur Sparc Station 2. Quelques remarques: tous les quotients successifs sont 1, ce qui signifie que l'algorithme de Lehmer s'applique de façon optimale. On peut en déduire, que, en moyenne les deux algorithmes, celui de SISYPHE et celui de BigNum qui repose sur l'arithmétique de [3] sont équivalents. L'avant-dernier calcul dans la version de Lisp ayant pris 5 heures de CPU, on peut estimer que le dernier calcul prend près de 7 heures. Il n'a pas été effectué.

n	sisyphe	BN	Lisp	AKCL	Maple	Pari
1000	0.05	0.55	41	2.2	0.06	0.08
2000	0.12	1.10	267	9.8	0.23	0.28
3000	0.20	1.71	588	20.5	0.50	0.59
4000	0.28	2.49	1845	37.3	0.90	1.01
5000	0.39	3.09	2927	59.0	1.37	1.56
6000	0.52	3.87	4073	84.6	1.98	2.24
7000	0.65	5.40	8654	115.3	2.67	3.05
8000	0.78	5.58	13364	151.4	3.50	3.94
9000	0.94	6.44	18000	193.0	4.44	4.93
9999	1.14	7.46		235.1	5.44	5.92

2.9 Bezout

Préliminaires

Étant donné deux entiers A et B, le but de cet algorithme est de trouver deux entiers u et v tels que $uA + vB = p$ où p est le pgcd de A et B. Nous supposerons ici $A > B > 0$. La fonction utilisateur ne fait bien entendu aucune hypothèse sur A et B.

Le calcul de la relation de Bezout implique le calcul du pgcd de A et B, et est une généralisation de l'algorithme d'Euclide. Posons $a_0 = A$, $b_0 = B$, et écrivons par récurrence $a_n = b_n q_n + r_n$, puis $a_{n+1} = b_n$, $b_{n+1} = r_n$, comme lors du calcul du pgcd.

Introduisons la matrice $Q_n = \begin{pmatrix} q_n & 1 \\ 1 & 0 \end{pmatrix}$, et quatre quantités x_n, y_n, u_n et v_n. Posons

$$\overline{M}_n = \begin{pmatrix} v_n & y_n \\ u_n & x_n \end{pmatrix} \qquad C_n = \begin{pmatrix} a_n \\ b_n \end{pmatrix} \tag{1}$$

avec $\overline{M}_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ comme condition initiale pour $n = 0$ et $C_0 = \begin{pmatrix} A \\ B \end{pmatrix}$.

La relation de récurrence donnant les a_n et les b_n s'écrit simplement

$$Q_n C_{n+1} = C_n. \tag{2}$$

Les quantités x_n, y_n, u_n et v_n sont définies par les relations $x_{n+1} = u_n$, $y_{n+1} = v_n$, $u_{n+1} = x_n + q_n u_n$, $v_{n+1} = y_n + q_n v_n$. Ce sont des quantités positives ou nulles. Les formules s'écrivent aussi sous la forme matricielle

$$\overline{M}_{n+1} = \overline{M}_n Q_n. \tag{3}$$

En combinant les relations (2) et (3) on voit que la matrice $\overline{M}_n C_n$ ne dépend pas de n. On en déduit

$$a_n v_n + b_n y_n = A \tag{4.a}$$

$$a_n u_n + b_n x_n = B \tag{4.b}$$

d'où l'on déduit que le pgcd de a_n et b_n divise le pgcd de A et B.

Comme la matrice Q_n est de déterminant -1, on a $x_n v_n - y_n u_n = (-1)^n$ donc l'inverse de $\overline{M}_n$ est $M_n = (-1)^n \begin{pmatrix} x_n & -y_n \\ -u_n & v_n \end{pmatrix}$. Les relations (4) s'écrivent donc aussi

$$(-1)^n x_n A - (-1)^n y_n B = a_n \tag{5.a}$$

$$-(-1)^n u_n A + (-1)^n v_n B = b_n \tag{5.b}$$

ou plus simplement $M_n C_0 = C_n$.

Une conséquence triviale de ces relations est que le pgcd de A et B divise le pgcd de a_n et b_n, donc lui est égal. Considérons M tel que b_M divise a_M. C'est le pgcd de A et B. Alors (5.b) s'écrit

$$-(-1)^M u_M A + (-1)^M v_M B = \mathrm{pgcd}(A, B). \tag{6}$$

Cette relation est appelée relation de Bezout, et l'algorithme que nous allons donner va calculer les différentes quantités. Nous allons faire cependant deux optimisations importantes.

Optimisations

Supposons que $M' = (-1)^m \begin{pmatrix} \alpha & -\beta \\ -\gamma & \delta \end{pmatrix}$ soit une matrice de déterminant $(-1)^m$ et supposons $C_{n+m} = M' C_n$ i.e.

$$a_{n+m} = (-1)^m a_n \alpha - (-1)^m b_n \beta \tag{7.a}$$

$$b_{n+m} = -(-1)^m a_n \gamma + (-1)^m b_n \delta \tag{7.b}$$

$$\alpha\delta - \beta\gamma = (-1)^m. \tag{7.c}$$

On a ce genre de relations si on utilise l'algorithme de Lehmer, ou si on applique l'algorithme de Bezout à a_n et b_n. De la relation $C_{n+m} = M' C_n$ on tire $M_{n+m} = M' M_n$. En d'autres termes:

$$x_{n+m} = \alpha x_n + \beta u_n \tag{8.a}$$

$$y_{n+m} = \alpha y_n + \beta v_n \tag{8.b}$$

$$u_{n+m} = \gamma x_n + \delta u_n \tag{8.c}$$

$$v_{n+m} = \gamma y_n + \delta v_n. \tag{8.d}$$

Estimons maintenant les quantités x, y, u et v. Comme $a_0 > b_0$, et $a_{n+1} > b_{n+1}$ (définition de la division euclidienne), on aura $q_n \geq 1$ pour tout n. On en déduit $v_{n+1} \geq v_n$. En particulier $v_n \geq 1$, d'où $y_n \geq 1$ pour $n > 0$, donc $v_{n+1} > v_n$ pour $n > 0$. En évaluant (4.a) en $n = M$, on voit que la suite des y_i et v_i est majorée par A/p donc par A si p est le pgcd cherché. De la même manière la suite des x_n est croissante (à partir de $n = 1$, car $x_0 = 1$ et $x_1 = 0$).

La seconde amélioration de l'algorithme consiste à ne calculer qu'une seule des deux quantités u_M, v_M, de préférence la plus petite u_M et à calculer l'autre via la relation (6). En fait, ceci n'est pas tout à fait évident. Nous avons constaté sur un exemple que la division finale prenait environ la moitié du temps de calcul de toutes les diverses quantités y_n et v_n. Calculer v_M par la boucle prend plus de temps (on travaille sur des nombres plus grands), par contre, la division finale se fait sur des nombres plus petits.

Détails de l'algorithme

Considérons d'abord les relations (8). Il s'agit de calculer $x_1 = \alpha x + \beta u$ et $u_1 = \gamma x + \delta u$, sachant que x et u sont des vecteurs de chiffres. On suppose que le premier indice de x est i_D et celui de u est i_E. Pour simplifier, on suppose que les deux vecteurs ont même taille. Cette taille sera dans la variable t_A, ce sera aussi la taille de a_n et b_n (cette façon de faire minimise le nombre de variables globales utilisées). On suppose également qu'un vecteur z peut être utilisé pour ranger des résultats intermédiaires. Évidemment, on veut mettre x_1 dans x et u_1 dans u en parallèle.

Notons que $x = (x_1 - \beta u)/\alpha$, donc $u_1 = (\gamma x_1 + (\alpha\delta - \beta\gamma)u)/\alpha$. Comme $\alpha\delta - \beta\gamma = \pm 1$, ceci se réduit à $u_1 = (\gamma x_1 \pm u)/\alpha$. On peut donc se passer de variable intermédiaire. Nous estimons cependant que la division est moins efficace que la multiplication, et qu'il est relativement plus efficace de calculer x_1 dans z, puis u_1 dans u et de copier z dans x. La différence de temps ne doit pas être en fait tellement grande.

La procédure `ibezout_add` a pour but de calculer $y = \alpha x + \beta u$. Comme $\alpha\delta - \beta\gamma = \pm 1$, les deux quantités α et β ne peuvent être simultanément nulles. On traite à part le cas où l'une des deux est nulle (en fait, l'autre ne peut être que 1 dans ce cas, mais cette procédure peut être appelée par ailleurs). On traite également à part le cas où l'une des deux quantités est 1, car le principe est le même que pour `itimesc0`.

Considérons le cas général. Pour chaque i on calcule $x_i\alpha + u_i\beta + c$ sous la forme $Ec' + z_i$ où c est la retenue entrante et c' la retenue sortante. Notons que $0 \leq c \leq 2E - 3$, et toutes les valeurs sont permises. Or $2E - 3 < E$ uniquement si $E = 2$, cas qui ne nous intéresse pas. Il faut donc gérer deux retenues. Pour cela, on écrit $x_i\alpha + c + \mathtt{C} = \mathtt{C}'E + p$, puis $\beta u_i + p = c'E + z_i$. À la fin de la boucle, il suffit d'additionner les deux retenues.

Relation de Bezout entre des chiffres

On calcule U, V, n et p tels que $-Ux + Vy = (-1)^n p$, où p est le pgcd de x et y. De plus M est incrémenté de n. Notons x_1 et y_1 les valeurs initiales de x et y. Supposons $u' = 0$ et $x' = 1$, de sorte que y_1 divise $x'x_1 - (-1)^n x$ et $u'x_1 + (-1)^n y$, au moins pour $n = 0$. Ces deux relations de divisibilité forment l'invariant de l'algorithme. Tant que y ne divise pas x, on écrit $x = qy + r$, on remplace x par y, y par r, et on incrémente n. On calcule les nouvelles valeurs de x' et u'. À la fin de la boucle, le pgcd est y. On sait que $-u'x_1 + Vy_1 = (-1)^n y$ pour un certain $V \geq 0$. Le calcul de V dépend de la parité de n.

Cas d'un nombre a et d'un chiffre b

Écrivons $a = bq + r$. Dans le cas où le reste est nul, le pgcd est b, $M = 0$ et $U = 0$, c'est un cas trivial. Dans le cas contraire, écrivons $-Ub + Vr = (-1)^n p$ grâce à la méthode proposée précédemment. On a donc $aV - b(U + qV) = (-1)^n p$. Il n'y a donc pas de calculs supplémentaires à faire.

Cas de deux nombres u et v

Sauf cas particulier, les deux quantités U et X auront à la fin à peu près la même taille que u et v, ce seront des vecteurs de chiffres. On commence par copier u et v dans des vecteurs de

taille t, la taille de u plus 2 (comme pour le calcul du pgcd). On alloue deux vecteurs de chiffres de même taille U et X, et un vecteur auxiliaire Z. Celui-ci servira de tampon à la fois pour la division et pour l'addition. On utilisera i_A et t_A comme indices dans u, i_B et t_B comme indices dans v. On supposera de plus que $t_A = t_B$ est le dernier indice utile de U et X. Le premier indice utile de ces deux derniers vecteurs y sera en position 0.

Les relations (8) sont calculées via `ibezout_add`, fonction qui calcule $\alpha X + \beta U$ dans X, U ou Z. On suppose que les chiffres précédant i_D ou i_E dans X ou U sont nuls. Si on note par A, B les valeurs initiales de u et v l'invariant de l'algorithme est que B divise $AX-(-1)^n u$ et $AU+(-1)^n v$.

Dans une première phase on appelle `gcd_via_bezout`. Cette procédure rend α, β, γ, δ, s et m, et on applique les relations (8). Il y a trois cas à considérer: $m = 0$, $m = 1$ et $m > 0$. Dans le premier cas, une seule division a été faite, par `quomod`. Il faut remplacer X par U et U par $X + Uq$. On utilise `itimes0` sachant que le vecteur u contient à la fois le quotient et le reste. Dans le cas $m = 1$, une seule division a été faite, et le quotient est δ. On utilise `ibezout_add` pour faire la multiplication. Dans les autres cas, $m \geq 2$, et il faut appeler `ibezout_add` deux fois. La quantité M est incrémentée de m (de 1 si $m = 0$), mais n'est incrémentée que de $m - 1$ dans le cas où le dernier reste est nul.

Dans ce cas de figure, on rend la partie utile de v, et on met dans U la partie utile de X. Dans le cas où le reste a au moins deux chiffres, on itère `bezout2`. Il reste le cas où v a un seul chiffre et u a plusieurs chiffres. Dans le cas où u a plus d'un chiffre, on fait une première division, en utilisant `iquoc`. Si la division est exacte, on remplace U par sa partie utile et on rend le chiffre de u. Dans le cas contraire, on fait une itération comme dans le cas $m = 1$. Finalement on est ramené au cas où u et v ont un chiffre, on appelle `ibezout0` et on calcule U en utilisant l'une des relations (8).

Algorithme 15. (Relation de Bezout) *On utilise des nombres internes a, b, p, des chiffres x, y, x_1, y_1, x', q, r, r_1, r_2, α, β, γ, δ, V, des indices i, j, k, s, n, des tableaux de chiffres u', v', z, X', U', Z', des vecteurs de chiffres Z, U, X, A. Les variables X, U, Z sont globales, X', U' et Z' désignent leurs pointeur de tas. La variable V est aussi globale. La variable globale M est un entier, la fonction principale rend le pgcd via la valeur de retour, et les globales U et M. On utilise également une variable globale P, vecteur de travail pour Karatsuba.*

Procédure nbezout, fonction principale. *Données a, et b* [On suppose $a > b > 0$].

1. *Poser $M = 0$.*

2.a. *Si a est un chiffre, rendre* `ibezout0(Int_val(`a`), Int_val(`b`))`.

2.b. *Si b est un chiffre*

Calculer `iquoc0(`a`, Int_val(`b`))`.

Si `C` $= 0$, *poser $U = 0$ et rendre b.*

Sinon, poser $p =$ `ibezout0(Int_val(`b`), C)`, *incrémenter M, poser $U =$* `make_int(`V`)` *et rendre p.*

2.c. *Cas général.*

Soit $s = t(a) + 2$, poser $i_A = 2$, $i_B = s - t(B)$. Allouer des vecteurs U, X, Z et P de taille s. Poser $t_A = t_B = s - 1$, $i_E = s$, $i_D = s - 1$.

Copier a et b dans des vecteurs de taille s, cadrés à droite.

Mettre 1 en position $s - 1$ dans X.

Rendre `ibezout2`(a,b).

Procédure ibezout0. *Données x et y* [calcule et rend le pgcd p, de plus U et V tels que $-Ux+Vy=(-1)^m p$, met à jour M].

1. *Poser* $n=0$, `C` $=0$, $x_1=x$, $y_1=y$, $u'=0$, $x'=1$.

2. *Faire* [boucle de division]

Poser $q=$ `ex/`(x,y), $r=$ `C`.

Si $r=0$, *fin de la boucle.*

Poser $x=y$, $y=r$, `C` $=0$, $r=$ `ex*`(q,u',x'), $x'=u'$ *et* $u'=r$.

Incrémenter n.

3. *Poser* $M=M+n$.

4.a. *Si n est impair, poser* `C` $=1$, $q=$ `ex*`$(u',x_1,$`ex-`$(y))$, *décrémenter* `C`, *et poser* $V=$ `ex/`(q,y_1).

4.b. *Sinon, poser* $V=$ `ex/`$($`ex*`$(u',x_1,y),y_1)$.

5. *Poser* $U=$ `make_int`(u'), *et rendre* `make_int`(y).

Macro ibezout_c. *Pas d'arguments* [Implémente le cas $m=0$, multiplication via `itimes0`. On utilise la variable globale P. Modifie les variables globales i_B, i_C, i_D, i_E et t_C.]

1. *Poser* $A=U$.

2. *Poser* $k=i_E$, $i_B=\min(i_D,i_E)$, $t_C=t_B$.

3. *Calculer* `itimes0`$(U',$`UHEAP`$(u),X',P')$.

4. *Poser* $U=X$, $X=A$.

5. *Poser* $i_E=i_C$ *et* $i_D=k$.

Procédure ibezout2. *Paramètres u et v.*

1. *Appeler* `ibezout3` *sur* `UHEAP`(u) *et* `UHEAP`(v), *échanger u et v.*

2.a. *Si* $i_B<t_B$, *rendre* `ibezout2`(u,v).

2.b. *Si* $i_B>t_B$ *remplacer U par* `nunderflow4`(X,i_D,t_B),
rendre la quantité `nunderflow4`(u,i_A,t_A).

2.c.1. *Sinon, poser* $r_1=v[i_B]$ [seul chiffre de v] *et* $r_2=u[i_A]$ [premier chiffre de u].

2.c.2. *Si* $i_A<t_A$ [u a plus d'un chiffre]

Poser $i_B=i_A$, *calculer* `iquoc`$($`UHEAP`$(u),r_1)$.

Si `C` $=0$ *remplacer U par* `nunderflow3`(U,i_E,t_B) *et rendre* `make_int`(r_1).

Sinon incrémenter M, poser $r_2=r_1$ *et* $r_1=$ `C`.

Poser $i_A=i_B$, $t_A=t_B$.

Appeler `ibezout_c`.

2.c.3 [On a donc deux fois un chiffre]

Sauver `UHEAP`(U) *dans* U'.

Poser $p=$ `ibezout0`(r_2,r_1).

Poser $k=$ `ibezout_add`$(X',$`Int_val`$(U),U',V,Z')$.

Poser $U =$ `nunderflow3`(Z, k, t_B).
Rendre p.

Procédure ibezout3. *Paramètres* u' *et* v' [On divise via Lehmer].

1. *Appeler* `gcd_via_bezout` *sur* u', v' *et 1. Ceci rend* α, β, γ, δ, s *et* N.

2. *Si* $N = 0$,

2.1. *Si* $i_B > t_B$ *ne rien faire.*

2.2. *Poser* $i = i_B$, $j = i_A$, $i_A = 0$, $t_A = i_C$.

2.3. *Appeler* `ibezout_c`.

2.4. *Poser* $i_B = i$, $i_A = j$, $t_A = t_B$.

2.5. *Incrémenter* M.

3. *Si* $N = 1$,

3.1. *Si* $i_B > t_B$ *ne rien faire.*

3.2. *Poser* $A = U$, $k = i_E$.

3.3. *Poser* $i_E =$ `ibezout_add`$(X', 1, U', \delta, X')$.

3.4. *Poser* $U = X$, $X = A$, $i_D = k$.

3.5. *Incrémenter* M.

4. *Sinon,* $[N \geq 2]$

4.1. *Poser* $k =$ `ibezout_add`$(X', \alpha, U', \beta, Z')$.

4.2. *Poser* $i_E =$ `ibezout_add`$(X', \gamma, U', \delta, U')$.

4.3. *Copier* Z *dans* X, *via* `bltvector`(X, k, Z, k), *poser* $i_D = k$.

4.4. *Incrémenter* M *de* N.

4.5. *Si* $i_B > t_B$, *décrémenter* M.

Procédure ibezout_add. *Paramètres* u', x, v', y *et* z [Calcule $u'x + v'y$, résultat dans z, entre les indices $\min(i_D, i_E)$ et t_A, copies via `ibltvector`$(z, 0, v, 0, t_A + 1)$].

1. *Poser* `C` $= 0$, $j = \min(i_D, i_E)$.

2.a. *Cas* $x = 0, y = 1$. *Si* $z \neq v'$, *copier* v' *dans* z, *poser* $j = i_E$.

2.b. *Cas* $x = 1, y = 0$. *Si* $z \neq u'$, *copier* u' *dans* z, *poser* $j = i_D$.

2.c. *Si* $x = y = 1$, *pour* i *de* t_A *à* j, *remplacer* z_i *par* `ex+`(u'_i, v'_i).

2.d. *Si* $x = 1$ *pour* i *de* t_A *à* j, *remplacer* z_i *par* `ex*`(v'_i, y, u'_i).

2.e. *Si* $y = 1$ *pour* i *de* t_A *à* j, *remplacer* z_i *par* `ex*`(u'_i, x, v'_i).

2.f. *Sinon, poser* $c = 0$. *Pour* i *de* t_A *à* j, *poser* $q =$ `ex*`(u'_i, x, c), $c =$ `C`, `C` $= 0$, *remplacer* z_i *par* `ex*`(v'_i, y, q).

2.g. [Cas général, suite] *Si* $c \neq 0$ *: si* `C` $= 0$, *poser* `C` $= c$, *sinon, poser* $c = c +$ `C` *; si overflow mettre* `C` *à 1 sinon à 0. Décrémenter* j, *mettre* c *dans* z_j.

3.a. *Si* `C` $\neq 0$, *décrémenter* j, *remplacer* z_j *par* `C`, *et rendre* j.

3.b *Sinon, si* z_j *est non nul rendre* j *sinon* $j + 1$.

Chapitre 3

Arithmétique rationnelle

3.1 Structures de données

Dans ce chapitre nous définissons les entiers et les rationnels et les opérations sur ces objets.

Un entier est soit un petit entier soit un grand entier. Dans le premier cas, il s'agit d'une structure C, créée par `make_int`, contenant un entier C, accessible par `Int_val`. Dans le chapitre précédent, cet entier C était considéré comme non signé, donc $0 \leq c < E$. On le considère signé, donc $-E/2 < c < E/2$. Dans la plupart des implémentations, le passage entre signé et non signé ne change pas la suite des bits. On fait l'hypothèse que si $0 \leq c < E/2$, où c est non signé, alors c considéré comme signé a exactement la même représentation. Si cette hypothèse est fausse, il faut changer un peu le code.

Un grand entier est un vecteur de type `#:r:z` à deux champs. Le premier champ est le signe (vrai si le nombre est positif, faux sinon) et le second est un nombre interne, en général un vecteur, sauf dans le cas où la valeur x du nombre est $E/2 \leq |x| < E$, cas où le champ est un petit entier interne, non signé.

Un nombre rationnel n'est jamais un entier, c'est le quotient de deux entiers a et b, représenté sous forme d'un vecteur de type `#:r:q` à quatre champs. Le premier champ est le signe. Il est suivi par les représentations internes des valeurs absolues de a et b. Finalement le dernier champ est un indicateur, qui s'il est vrai indique que a et b sont premiers entre eux.

Par défaut, toutes les fractions sont systématiquement réduites. La variable-fonction **fractions-are-reduced** permet de consulter ou modifier un indicateur interne `simp_flag`. Si celui-ci est faux, le calcul de pgcd n'est pas systématique. On peut dans ce cas réduire une fraction individuelle à l'aide de la fonction **reduce-fraction**. Cette fonction rend une nouvelle fraction, mais elle peut aussi modifier physiquement la fraction si la variable interne associée à la variable fonction **reduced-fractions-are-displaced** est vraie.

La fonction (interne) principale de création des fractions est `qx`. Elle prend en arguments les valeurs absolues des numérateurs et dénominateurs. Le signe de la fraction est dans la variable globale `sign1`. De plus, elle suppose positionné l'indicateur `is_simplified` : si la valeur est vraie, le numérateur et le dénominateur sont premiers entre eux. Dans le cas contraire, il faut calculer un pgcd. Ce pgcd n'est calculé que si la variable `simp_flag` est vraie. Si elle est fausse, on ne

fait pas de calcul de pgcd. Il faut par contre s'assurer que le résultat n'est pas entier. Pour être plus efficace, la fonction suppose que la variable `is_not_integer` est positionnée: si elle est vraie, la fraction est garantie ne pas être un entier, si elle est fausse, on divise le numérateur par le dénominateur pour s'en assurer. Il y a plusieurs fonctions de création des entiers: par exemple `zx` crée un entier en faisant les mêmes hypothèses que `qx`, la fonction `mknumpos` crée un entier positif, etc.

L'implémentation des variables/fonctions dans notre version de Lisp est la suivante: dans le cas simple comme `ibase`, où la valeur est un petit entier, la valeur se trouve dans un tableau `Istack`, à l'indice `IS_ibase`. L'objet Lisp **ibase** a une valeur fonctionnelle, qui est une fonction à 0 ou 1 argument. La fonction C associée reçoit le nombre d'arguments n (un entier C) et un objet Lisp x qui est l'argument, ou une valeur bidon. La différence entre une fonction à 0 ou 1 arguments est que `(setq ibase 10)` est interprété comme `(ibase 10)`. Une autre différence est que `(let ((ibase 10)) ...)` utilise un mécanisme spécial pour ranger l'ancienne valeur de la base. En fait, la macro `push_internal` fait le travail, et la fonction `ibase` est appelée avec $n = 2$. Grâce à ce mécanisme, les variables fonctions définies plus loin, qui prennent en argument des booléens Lisp, peuvent rentrer dans ce cadre (la conversion booléens Lisp/booléens C est faite par la fonction, et non par la fonction `let`).

Algorithme 16. (Primitives pour les nombres) *On utilise un objet Lisp x, des entiers internes a, b, c et p, et des entiers C, n et m. On utilise comme variables globales* `simp_flag`, `displace_flag`, `complex_flag`, *qui sont les objets dans* `Istack`, *à la position i, indice qui est la valeur de la macro C obtenue en préfixant la variable par* `IS_`.

Procédure my_get_set. *Arguments n, x, i* [x objet Lisp, n et i entiers C].

1. *Si $n > 0$, poser $m = 0$ si x est faux, $m = 1$ sinon.*
2. *Si $n = 2$, appeler* `push_internal`(i, m).
3. *Si $n = 1$, mettre m dans* `Istack` *en position i.*
4. *Si* `Istack` *en position i contient 0, rendre faux, sinon rendre vrai.*

Fonction fractions-are-reduced. *Arguments n, x.*

Rendre `my_get_set`$(n, x,$ `IS_simp_flag`$)$.

Fonction reduced-fractions-are-displaced. *Arguments n, x.*

Rendre `my_get_set`$(n, x,$ `IS_displace_flag`$)$.

Fonction print-complex-as-common-lisp. *Arguments n, x.*

Rendre `my_get_set`$(n, x,$ `IS_complex_flag`$)$.

Fonction reduce-fraction. *Argument x.*

Si x n'est pas une fraction rationnelle, rendre x.

Rendre `resimp0`(x).

Procédure qx. *Paramètres a, b* [construit a/b].

Si $b = 0$, erreur, division par 0.

Si $b = 1$, *rendre* `zx`(a).

Sinon rendre `qx0`(a, b).

Macro pgcd_hack. *Arguments* a, b [divise a et b par leur pgcd].

Soit $p =$ `npgcd`(a, b).

Si $p = 1$, *ne rien faire.*

Sinon diviser a *et* b *par* p *via* `nquomod`.

Procédure qx0. *Arguments* a *et* b, *des entiers internes.*

Si $a = 0$, *rendre* 0.

Si $a = 1$, *ou si* `is_simplified` *est vrai, rendre* `mkrat`$(a, b, vrai)$.

Si `simp_flag` *est vrai*

Mettre `is_simplified` *à vrai.*
Appeler `pgcd_hack` *sur* a *et* b.
Rendre `qx`(a, b).

Si `is_not_integer` *est vrai, rendre* `mkrat`$(a, b, faux)$.

Soit c *le quotient de* a *par* b *calculé via* `nquomod`. *Si le reste de la division est nul rendre* `zx`(c), *sinon* `mkrat`$(a, b, faux)$.

Procédure zx. *Argument* a.

Si a *est un entier*

Poser $n =$ `Int_val`(a).
Si $n < 0$, *rendre* `mkbignum`(a).
Si `sign1` *est vrai, rendre* a, *sinon,* `make_int`$(-n)$.

Sinon rendre `mkbignum`(a).

Procédure nx. *Argument* a.

Mettre `sign1` *à vrai.*

Rendre `zx`(x).

Procédure displace. *Arguments* x *et* y.

Si x *et* y *sont des vecteurs de longueur 4, copier les 4 éléments de* y *dans* x *et rendre* x.

Sinon rendre y.

Procédure resimp. *Argument* x [x est une fraction].

Sauver `sign1` *dans* s.

Mettre le signe de x *dans* `sign1`, *récupérer* a, b *les numérateur et dénominateur de* x. *Mettre* `is_simplified` *à vrai.*

Appeler `pgcd_hack` *sur* a *et* b.

Soit $y =$ `qx`(a, b). *Remettre* `sign1` *à son ancienne valeur.*

Si `displace_flag` *est faux, rendre* y.

Sinon rendre `displace`(x, y).

Procédure resimp0. *Argument* x.

Si le flag réduit de x *est vrai rendre* x.

Sinon, rendre `resimp`(x).

3.2 Fonctions élémentaires sur les rationnels

Les deux fonctions génériques **numerator** et **denominator** rendent le numérateur et le dénominateur d'un nombre. Ces fonctions n'ont de sens que pour les nombres rationnels, en d'autres termes, provoquent une erreur d'intitulé (en anglais) « This operation has no meaning for its argument ». Le numérateur d'un entier est l'entier lui-même, le dénominateur est 1 par convention. En ce qui concerne une fraction a/b, le numérateur est la quantité a (avec le signe de la fraction) est le dénominateur est b (positif). Ces deux fonctions réduisent systématiquement les fractions, contrairement à l'option choisie dans LeLisp, mais conforme à celle de Common Lisp (cf [10, p. 215].

La fonction générique **0−** existe pour tous les nombres définis dans Sisyphe. En ce qui concerne les flottants et les petits entiers, il suffit de copier l'objet, en remplaçant la partie interne par son opposé. Pour les grands entiers, il suffit de changer le signe. Dans le cas d'une fraction, on fait de même, mais le résultat est réduit s'il le faut. On utilise donc `qx` plutôt que `mkrat`. Finalement, pour prendre l'opposé d'un nombre complexe, il suffit de prendre l'opposé des parties réelle et complexe.

La fonction générique **1/** calcule l'inverse d'un nombre. Notons que l'inverse de 0 n'existe pas, les entiers 1 et -1 sont leur propres inverses. Pour tous les autres entiers x, leur inverse est $1/x$, le résultat n'est certainement pas entier, et la fraction est toujours réduite. L'inverse d'une fraction a/b est la fraction b/a. Si la fraction de départ est réduite, l'inverse l'est aussi, et le résultat ne peut être entier que si a est trivial. Dans le cas où la fraction n'est pas réduite, on ne peut rien dire a priori. Finalement l'inverse d'un nombre complexe $a + ib$ est le nombre $a/d - ib/d$ où le dénominateur d est $a^2 + b^2$.

3.3 Comparaison

La fonction générique **signum** rend le signe de son argument. Pour les nombres complexes, elle est définie par la formule $x/|x|$, le résultat est donc un nombre complexe. Dans les autres cas, le résultat est 0 si le nombre est nul, 1 s'il est positif, et -1 sinon (notons que le résultat est toujours entier, contrairement à la stratégie préconisée par Common Lisp).

La fonction **abs** rend la valeur absolue d'un nombre. Ceci est simple dans le cas des nombres réels. Par convention la valeur absolue d'un nombre complexe $a + ib$ est son module $\sqrt{a^2 + b^2}$. La racine carrée est calculée en utilisant l'arithmétique générique.

Toutes les fonctions de comparaison que ce soient les fonctions d'égalité ou d'inégalité, utilisent `icompare`, qui rend 0, 1 ou -1 suivant que le premier nombre est égal, plus grand ou plus petit que le second. Cette fonction rend 2 si l'un des arguments n'est pas un nombre ou si on compare deux

nombres complexes différents. Les fonctions Lisp rendent le premier argument si la comparaison est vraie, faux sinon.

L'algorithme de comparaison est le suivant: si l'un des arguments n'est pas un nombre, on rend 2. Si le deux arguments sont complexes, on compare parties réelles et parties imaginaires. On rend 0 en cas d'égalité, 2 en cas d'inégalité. Si l'un des arguments est un flottant, on convertit les deux arguments en flottants, et on les compare. Dans les autres cas, on calcule le signe. Si les signes sont distincts, la comparaison est triviale. Si les signes sont identiques et négatifs, on échange les deux nombres. On est donc amené à comparer les valeurs absolues de deux entiers ou rationnels.

Comparaison de deux entiers a et b. Si les deux entiers sont des chiffres, on utilise la fonction de comparaison C. Si un seul est un chiffre, c'est le chiffre le plus petit. Dans le cas contraire, cas de deux vecteurs de chiffres, on utilise la fonction `ncmp`.

La situation se complique pour la comparaison d'un rationnel x et d'un nombre y, entier ou rationnel. On écrit $x = \pm a/b$, $y = \pm c/d$. Les champs signes sont ignorés, on suppose les nombres positifs. Si $a < c$ et $b > d$ alors $x < y$ (s'il y a deux égalités, on a $x = y$, et s'il y a une égalité on a $x < y$). De même, si $a > c$ et $b < d$ on a $x > y$. Par ailleurs, si $a < b$ et $c > d$, on a $x < 1 < y$ d'où $x < y$, et l'on peut renverser le sens des inégalités. Dans le cas où aucun de ces critères ne donne de résultat, on compare ad et bc. On évite ainsi au maximum les multiplications, de plus, les tests à zéro (tests de signe) sont triviaux.

Algorithme 17. (Comparaison des nombres) *On utilise les nombres x, y, des chiffres n, m, des indices s_x, s_y, des nombres internes a, b, c et d.*

Fonction <. *Arguments x et y.*

Si `icompare`(x, y) *est* -1 *rendre x sinon faux.*

Fonction >. *Arguments x et y.*

Si `icompare`(x, y) *est* 1 *rendre x sinon faux.*

Fonction <=. *Arguments x et y.*

Si `icompare`(x, y) *est* 0 *ou* -1 *rendre x sinon faux.*

Fonction >=. *Arguments x et y.*

Si `icompare`(x, y) *est* 0 *ou* 1 *rendre x sinon faux.*

Fonction =. *Arguments x et y.*

Si `icompare`(x, y) *est* 0 *rendre x sinon faux.*

Fonction <>. *Arguments x et y.*

Si `icompare`(x, y) *n'est pas* 0 *rendre x sinon faux.*

Fonction <?>. *Arguments x et y.*

Rendre `make_int`(`icompare`(x, y)).

Procédure isignum. *Argument x.*

Si x est un grand entier ou un rationnel, extraire son champ signe. Si vrai, rendre 1 sinon -1.

Si x est un petit entier, récupérer sa valeur interne via `Int_val` *la comparer à 0 et rendre le signe. Si x est un flottant, faire de même en utilisant* `Double_val`.

Dans les autres cas, rendre 2.

Fonction signum. *Argument x.*

Soit $n =$ `isignum`(x).

Si n est 0, 1 ou -1, rendre `make_int`(n).

Si x n'est pas un nombre complexe, erreur.

Sinon rendre $x/|x|$.

Fonction abs. *Argument x.*

Soit $s =$ `isignum`(x).

Si s est 0, 1 ou -1, rendre 0, x ou $-x$.

Si x n'est pas un nombre complexe, erreur.

Sinon $x = a + ib$, rendre $\sqrt{a^2 + b^2}$.

Procédure icompare. *Arguments x et y.*

1. *Calculer les types de x et y.*

2. *Si l'un des objets n'est pas un nombre, rendre 2.*

3. *Si les deux nombres ont la même adresse, rendre 0.*

4. *Si un et un seul des nombres est complexe, rendre 2.*

5. *Si les deux nombres sont complexes, comparer parties réelles et parties imaginaires. S'il y a égalité dans les deux cas, rendre 0 sinon 2.*

6. *Si l'un des deux nombres est flottant, convertir les deux en flottants, et les comparer.*

7. *Calculer dans s_x et s_y les signes de x et y (comme dans* `isignum`*). Si x est entier, mettre* `Int_val`(x) *dans n, si y est entier, mettre* `Int_val`(y) *dans m.*

8. *Si $y = 0$ rendre s_x, si $x = 0$ rendre $-s_y$.*

9. *Si $s_x \neq s_y$ rendre s_x.*

10. *Si $s_x = -1$ échanger x et y.*

11. *Si x est un petit entier, y un grand entier, rendre -1, si y est un petit entier, x un grand entier, rendre 1.*

12.a. *Si x est un petit entier: poser $b = 1$, si $x < 0$ poser $a =$* `make_int`$(-$`Int_val`$(x))$ *sinon poser $a = x$.*

12.b *Si x est un grand entier, mettre dans a la valeur absolue de x, poser $b = 1$.*

12.c. *Sinon, $x = \pm a/b$ est un rationnel. Mettre les champs adéquats dans a et b.*

12.d *Écrire de même $y = \pm c/d$.*

13. *Si x et y sont des entiers, rendre* `ncmp`(a, c).

14. *Poser* $t_x = \texttt{ncmp}(a, c)$, $t_y = \texttt{ncmp}(d, b)$.
Si $t_x = 0$ *rendre* t_y.
Si $t_y = 0$ *rendre* t_x.
Si $t_x = t_y$ *rendre* t_x.

15. *Poser* $t_x = \texttt{ncmp}(a, b)$, $t_y = \texttt{ncmp}(d, c)$. *Si* $t_x = t_y$ *rendre* t_x.

16. *Rendre* $\texttt{ncmp}(ad, bc)$ [produit via `ntimes`].

Procédure ncmp. *Arguments a et b, entiers internes.*

1. *Si a et b sont des chiffres, comparer* $\texttt{Int\_val}(a)$ *et* $\texttt{Int\_val}(b)$ *via* `ex?`.

2. *Si a est un chiffre, rendre* -1.

3. *Si b est un chiffre, rendre* 1.

4. *Si a et b ont le même adresse, rendre* 0.

5. *Si a et b n'ont pas la même taille, rendre* 1 *si a est le plus long,* -1 *sinon.*

6. *Comparer les chiffres de a et b de gauche à droite. S'ils sont tous égaux, rendre* 0.

7. *Soit i le premier indice pour lequel* $a_i \neq b_i$. *Si* $a_i < b_i$, *rendre* -1, *sinon* 1.

3.4 Addition et multiplication

Les deux fonctions qui additionnent ou multiplient des nombres sont des fonctions génériques de Lisp, à nombre arbitraire d'arguments. En règle générale, elles utilisent un résultat partiel, et ajoutent chaque nouvel élément à ce résultat partiel. On se ramène ainsi à une opération binaire.

Examinons d'abord les cas les plus simples : pour additionner (ou multiplier) deux nombres complexes $x = a + ib$ et $y = c + id$, on utilise les formules $x + y = (a + c) + i(b + d)$ et $xy = (ac - bd) + i(ad + bc)$, et l'arithmétique générique pour faire les calculs. Si un seul des deux nombres est complexe, par commutativité, on peut supposer que c'est x, et on applique les relations précédentes avec $c = y$ et $d = 0$. Si l'un des arguments est un flottant, l'autre argument est converti en flottant.

Nous considérons maintenant le cas de la somme ou produit de deux nombres, qui sont des chiffres, des entiers ou des fractions, en traitant d'abord la somme, puis le produit. On peut évidemment supposer les deux nombres non nuls (0 est élément neutre pour la somme et absorbant pour le produit). Dans le cas où `simp_flag` est vrai, le résultat est réduit. Les opérations $0 + x$, $x - 0$, etc., ne sont donc pas tout-à-fait triviales.

Si on veut additionner deux petits entiers, la méthode est la suivante : si les nombres ont des signes distincts, le résultat est un petit entier, il ne peut y avoir de dépassement de capacité. Dans le cas contraire, soient x et y les valeurs à ajouter. Quitte à changer le signe, on peut supposer qu'elles sont positives, donc $x < E/2$ et $y < E/2$. La somme est $x + y < E$, et se calcule en non signé. Si $x + y < E/2$, on est dans le cas précédent, et dans le cas contraire, on a un grand entier, dont la valeur absolue est un chiffre. Pour simplifier le code, le calcul de $x - y$ consiste à ajouter l'opposé de y à x, sauf si x et y ont le même signe, ou ont des valeurs absolues < 10000, cas où le résultat est garanti tenir sur un chiffre (la constante 10000 pourrait être remplacé par $E/4$).

Dans les autres cas de figure, on extrait les signes de x et y, et on les met dans des variables globales. Dans le cas de la somme de deux entiers, petits ou grands, à l'exclusion du cas précédent,

on calcule les valeurs absolues. Ceci est trivial pour un grand entier, mais peut allouer de la mémoire dans le cas contraire. On utilise `rqspll`, fonction qui somme les deux valeurs absolues avec les signes adéquats, qui met le signe dans `sign1` et qui rend la valeur absolue du résultat. Cette procédure est également utilisée dans la suite.

Il reste alors le cas de la somme de deux fractions, ou d'une fraction et d'un entier. Si on veut un résultat réduit, on commence par réduire les fractions qui apparaissent dans la somme. On suppose $x = \pm a/b$. Si y est entier, le résultat est $(\pm a + by)/b$. Si b est premier à a, cette fraction est réduite ; le résultat n'est pas entier.

Finalement, il reste à sommer $x = \pm a/b$ et $y = \pm c/d$. Si les dénominateurs sont les mêmes, on calcule simplement $(\pm a + \pm c)/d$. Si on ne veut pas un résultat réduit, on calcule $(\pm ad + \pm bc)/bd$. Sinon, soit p le pgcd de b et d, $b' = b/p$, $d' = d/p$. Le résultat est $(\pm ad' + \pm b'c)/b'd'p$. On suppose a/b et c/d réduit, de sorte que b' est premier à ad'. La seule simplification qui peut encore se produire est entre p et le numérateur. Le cas $p = 1$ est donc trivial. Notons aussi que le numérateur n'est jamais nul, et le résultat n'est jamais entier (on a exclus le cas $b = d$).

Algorithme 18. (Addition générique) *Les variables et paramètres utilisés ici sont x et y des nombres, a, b, c, d, N, D, p, X et Y des entiers internes,* `is_simplified`, `is_not_integer`, `sign1` *et* `sign2` *des variables globales booléennes.*

Procédure rqspll. *Paramètres X et Y* [Rend la valeur absolue de la somme de X et Y, les signes sont dans `sign1` et `sign2`, le signe du résultat est dans `sign1`].

1. *Si* `sign2` *et* `sign1` *sont égaux, rendre* `nadd`(X, Y).

2. *Soit* $s =$ `ncmp`(X, Y).

2.a. *Si $s = 0$, rendre 0.*

2.b *Si $s = 1$, rendre* `ndiff`(X, Y).

2.c *Sinon remplacer* `sign1` *par* `sign2` *et rendre* `ndiff`(Y, X).

Fonction add. *Argument x et y.*

1. *Calculer les types de x et y.*

2. *Erreur si l'un des deux objets n'est pas un nombre.*

3. *Si x est l'entier 0, si* `simp_flag` *est vrai, rendre* `reduce-fraction`(y)*, sinon y. Idem si $y = 0$.*

4. *Si $x = a + ib$ est complexe, si y est réel, rendre $(a + y) + ib$, et si $y = c + id$ est complexe, rendre $(a + c) + i(b + d)$. Idem si seul y est complexe.*

5. *Si x ou y est un flottant, convertir les deux arguments en flottants, rendre la somme en tant que flottant.*

6. *Si x et y sont des petits entiers*

6.1. *Soit $n =$* `Int_val`$(x), m =$ `Int_val`(y).

6.2. *Si n et m ont des signes distincts, rendre* `make_int`$(n + m)$.

6.3. *Si $n > 0$ et $m > 0$, poser $s = 1$, sinon, $s = -1$, changer le signe de n et m.*

6.4. *Soit $p = n + m$ (calcul, résultat sous forme de chiffres).*

6.5. *Si le bit de signe de p n'est pas positionné, rendre* `make_int`$(\pm p)$.

6.6. *Mettre dans* `sign1` *la valeur vraie si* $s = 1$, *faux sinon.*

6.7. *Rendre* `mkbignum`(`make_int`(p)).

7. *Si* x *est un petit entier, soit* $n =$ `Int_val`(x). *Mettre dans* `sign1` *le signe de* n. *Si* $n < 0$, *poser* $a =$ `make_int`($-n$), *sinon poser* $a = x$. *Si* x *est un grand entier, mettre dans* `sign1` *son signe, dans* a *sa valeur absolue. Dans les deux cas poser* $b = 1$. *Idem pour* y, *en utilisant* c, d, *et* `sign2`.

8. *Si ni* x *ni* y *ne sont des fractions, rendre* `zx`(`rqspll`(a, c)).

9. *Si* x *et* y *sont des fractions,*

9.1. *Si* `simp_flag` *est vrai, remplacer* x *par* `resimp0`(x) *et* y *par* `resimp0`(y).

9.2. *Récupérer les signes de* x *et* y, *comme dans les autres cas, et les champs* a, b, c, d, *de sorte que* $x = \pm a/b$ *et* $y = \pm c/d$.

9.3. *Si* `simp_flag` *est vrai*

9.3.1. *Si* $b = d$ *(via* `ncmp`*) mettre* `is_simplified`, `is_not_integer` *à faux, aller en* **9.4.5.**

9.3.2. *Mettre* `is_simplified` *à vrai, poser* $p =$ `npgcd`(b, d).

9.3.3. *Si* $p = 1$ *aller en* **9.4.3.**

9.3.4. *Diviser* b *et* d *par* p *via* `nquomod`.

9.3.5. *Poser* $a =$ `rqspll`(ad, bc), $b = bd$, *produits via* `ntimes`.

9.3.6. *Appeler* `pgcd_hack`(a, p).

9.3.7. *Rendre* `qx`(a, bp) *(produit via* `ntimes`*).*

9.4. *Sinon* [deux fractions, résultat non réduit]

9.4.1. *Mettre* `is_simplified`, `is_not_integer` *à faux.*

9.4.2. *Si* $b = d$ *(via* `ncmp`*), aller en* **9.4.5.**

9.4.3. *Poser* $a =$ `rqspll`(ad, bc), $b = bd$, *produits via* `ntimes`.

9.4.4. *Rendre* `qx`(a, b).

9.4.5. *Poser* $a =$ `rqspll`(a, c). *Rendre* `qx`(a, d).

10. *Si* y *est un nombre rationnel, mettre* `sign1` *dans* `sign2`, *poser* $c = a$, $x = y$.

11. *Si* `simp_flag` *est vrai, poser* $x =$ `resimp0`(x).

12. *Mettre faux dans* `is_not_integer`, *le flag* `rq_rflag` *de* x *dans* `is_simplified`.

13. *Mettre le signe de* x *dans* `sign1`, *écrire* $x = \pm a/b$.

14. *Rendre* `qx`(`rqspll`(a, bc), b).

Procédure add0. *Argument* x [utilisée par $x + 0$, $x - 0$, etc.].

Si x *n'est pas un nombre, signaler une erreur.*

Si `simp_flag` *est vrai, rendre* `reduce-fraction`(x).

Sinon rendre x.

Fonction 1+. *Argument* x [incrémente x].

1. *Si* x *n'est pas un petit entier, rendre* $x + 1$, *via* `add`.

2. *Soit* $n =$ `Int_val`(x).

3. *Si* $n \leq 1000$, *rendre* `make_int`($n + 1$).

4. *Sinon,* [on a $0 < n < E/2$] *additionner 1 à* n*, modulo* 2^{32}.
5. *Si* $n < E/2$*, rendre* `make_int`(n)*, sinon rendre* 2^{31}.

Fonction 1-. *Argument* x [décrémente x].

1. *Si* x *n'est pas un petit entier, rendre* $x - 1$*, via* `sub`.
2. *Soit* $n =$ `Int_val`(x).
3. *Si* $n \geq -1000$*, rendre* `make_int`$(n - 1)$.
4. *Sinon, soustraire 1 de* x*, modulo* 2^{32}.
5. *Si* $n \neq -E/2$*, rendre* `make_int`(n)*, sinon rendre* -2^{31}.

Fonction +. *Fonction à nombre variable* n *d'arguments.*

1. *Si* $n = 0$*, rendre 0.*
2. *Soit* x *le dernier argument.*
3. *Si* $n = 1$*, rendre* `add0`(x).
4. *Sinon, pour chaque autre argument* y*, remplacer* x *par* $x + y$.
5. *Rendre* x.

Fonction -. *Fonction à nombre variable* n *d'arguments.*

1. *Si* $n = 0$*, rendre 0.*
2. *Si* $n = 1$*, soit* x *le seul argument. Rendre* $-x$ *(opposé).*
3. *Sommer les* $n - 1$ *derniers arguments via la fonction '+'.*
4. *Soit* y *le résultat,* x *le premier argument.*
5. *Rendre* $x - y$ *(différence).*

Fonction 0-. *Argument* x*, calcule* $-x$.

1. *Si* x *est un petit entier, rendre* `make_int`$(-$`Int_val`$(x))$.
2. *Si* x *est un flottant, faire de même en flottant.*
3. *Si* x *est un grand entier, mettre dans* `sign1` *l'opposé du signe de* x*. Rendre* `zx`*(*`rz_num`*(*x*))*.
4. *Si* x *est une fraction, mettre dans* `sign1` *l'opposé du signe de* x*, dans* `is_simplified` *le flag réduit de* x*. Écrire* $x = a/b$*. Mettre* `is_not_integer` *à vrai. Rendre* `qx`(a, b).
5. *Si* x *est un nombre complexe, rendre le nombre complexe dont les parties réelle et imaginaire sont les opposées des parties réelle et imaginaire de* x.
6. *Erreur sinon.*

Fonction sub. *Argument* x *et* y [Soustraction binaire]

1. *Si* $y = 0$*,* x *un petit entier ou un flottant, rendre* x.
2. *Si* x, y *sont deux petits entiers de même signe, rendre* `make_int`$($`Int_val`$(x) -$ `Int_val`$(y))$.
3. *Si* x *et* y *sont deux petits entiers ou flottants (au moins un des deux est flottant) prendre les champs internes de* x *et* y*, les soustraire en flottant, rendre un flottant.*
4. *Dans tous les autres cas, ajouter l'opposé de* y *à* x.

Donnons maintenant l'algorithme du produit. On utilise une procédure `rqstimes1` qui prend en argument une fraction $x = \pm a/b$, deux entiers internes c et d. Cette fonction suppose que le signe du produit est dans `sign1` et qu'on veut un résultat réduit. Ce résultat est $(ac)/(bd)$. Les seuls pgcd à extraire sont entre a et d, et entre b et c.

La fonction principale de multiplication exclut d'abord les cas triviaux, i.e. multiplication par 0, 1 ou -1, et les cas plus compliqués, cas où l'un des nombres est complexe ou flottant. Il ne reste donc que le cas d'entiers ou de rationnels. La première chose à faire est de calculer le signe du produit. Le premier nombre est $x = \pm a/b$, le second est $\pm c/d$.

Dans le cas où les nombres sont deux petits entiers, on utilise une copie de `ntimes`. Dans ce cas a et c ne sont pas calculés, ceci pour éviter d'allouer de la mémoire, dans le cas où le nombre est négatif. Plutôt que de recalculer le signe, on préfère le mettre dans une variable locale. Le produit de grands entiers se fait évidemment par `ntimes`.

Pour multiplier une fraction $x = a/b$ par une fraction $y = c/d$, ou un entier $y = c$, cas où $d = 1$, on teste d'abord les cas triviaux : si $b = c$, le résultat est a/d, calculé via `qx`, et il s'agit d'un entier si $d = 1$. On teste de même $a = d$. Dans les autres cas, on rend ac/bd, calculé directement, ou via `rqstimes1`. Notons que si le résultat est réduit, on fait les tests avant de réduire x et y. Il est inutile de les faire après réduction, si $b = c$, extraire le pgcd est trivial.

Algorithme 19. (Multiplication générique) *Dans cet algorithme on utilise comme variables locales et paramètres, x et y des nombres, a, b, c, d, N, D, p, X et Y des entiers internes,* `is_simplified`, `is_not_integer`, `sign1` *et* `sign2` *des variables globales booléennes.*

Fonction *. *Fonction à nombre variable n d'arguments.*

1. *Si $n = 0$, rendre 1.*
2. *Soit x le dernier argument.*
3. *Si $n = 1$, rendre* `add0`(x).
4. *Sinon, pour chaque autre argument y, remplacer x par xy.*
5. *Rendre x.*

Fonction /. *Fonction à nombre variable n d'arguments.*

1. *Si $n = 0$, rendre 1.*
2. *Si $n = 1$, soit x le seul argument. Rendre $1/x$ (inverse).*
3. *Multiplier les $n - 1$ derniers arguments via la fonction $*$.*
4. *Soit y le résultat, x le premier argument.*
5. *Rendre x/y (quotient).*

Fonction 1/. *Argument x, calcule $1/x$.*

1. *Si x est un petit entier,*
   - 1.1. *Si $x = 0$, division par 0.*
   - 1.2. *Si $x = \pm 1$, rendre x.*
   - 1.3. *Positionner dans* `sign1` *le signe de x et si $x < 0$, remplacer x par $-x$.*
   - 1.4. *Rendre* `mkrat`$(1, x, vrai)$.

2. *Si x est un flottant, rendre l'inverse de x.*
3. *Si x est un grand entier, mettre dans* `sign1` *le signe de x. Rendre* `qx`$(1,$ `rz_num`$(x))$.
4. *Si x est une fraction, mettre dans* `sign1` *le signe de x, dans* `is_simplified` *le flag réduit de x. Écrire $x = a/b$. Mettre* `is_not_integer` *à faux. Rendre* `qx`(b, a).
5. *Si $x = a + ib$ est un nombre complexe, poser $t = 1/(a^2 + b^2)$, rendre le nombre complexe $at - ibt$.*
6. *Erreur sinon.*

Fonction div. *Argument x et y* [division binaire].

1. *Si $y = 1$, x un petit entier ou un flottant, rendre x.*
2. *Si $y = 0$, erreur.*
3. *Si x et y sont deux petits entiers,*

   Poser $a =$ `Int_val`(x), $b =$ `Int_val`(y).

   Si le reste de la division de a par b est nul, rendre `make_int`(a/b).
4. *Si x et y sont deux petits entiers ou flottants (au moins un des deux est flottant) prendre les champs internes de x et y, les diviser en flottant, rendre un flottant.*
5. *Dans tous les autres cas, multiplier l'inverse de y par x.*

Procédure rqstimes1. *Cette procédure prend en argument une fraction $x = a/b$ et deux entiers internes c et d. Elle rend cx/d réduit.* [Elle suppose que x et c/d sont des fractions réduites, les signes sont précalculés].

1. *Soit $x = \pm a/b$, positionner* `is_simplified` *à vrai.*
2. *Appeler* `pgcd_hack`(a, d), `pgcd_hack`(c, b).
4. *Calculer le numérateur $N =$* `ntimes`(a, c) *et le dénominateur $D =$* `ntimes`(b, d).
5. *Rendre* `qx`(N, D).

Fonction mul. *Arguments x et y.*

1. *Calculer les types de x et y. Erreur si ce ne sont pas des nombres.*
2. *Si $x = 0$, rendre 0, si $x = -1$, rendre $-y$, si $x = 1$, rendre* `add0`(y). *Mêmes tests sur y.*
3. *Si l'un des deux x ou y est complexe:*

   3.1. *Si $x = a + ib$, $y = c + id$, rendre $(ac - bd) + i(ad + bc)$.*

   3.2. *Si $x = a + ib$, y est réel, rendre $ay + iby$. Idem si x est réel.*
4. *Si l'un des deux x, y est flottant, convertir les deux nombres en flottant, calculer le produit et rendre un flottant.*
5. *Si x est un petit entier, mettre son signe dans* `sign1`*, sa valeur absolue (en tant que petit entier C) dans n. Si x est un grand entier, mettre son signe dans* `sign1`*, sa valeur absolue (en tant que entier interne) dans a. Sinon x est une fraction. Mettre son signe dans* `sign1`.
6. *Faire de même pour y (on utilise m, c et* `sign2`*). Poser $b = d = 1$.*
7. *Sauver les signes dans s_1 et s_2. S'ils sont égaux, mettre vrai sinon faux dans* `sign1`.

8. *Si x et y sont des petits entiers.*

8.1. *Poser* `C` $= 0, r =$ `ex*`$(n, m, 0)$.

8.2. *Si* `C` *n'est pas 0, allouer un vecteur de chiffres z de taille 2, y mettre* `C` *et r. Rendre* `zx`(z).

8.3. *Si le bit de signe de r est positionné, rendre* `mkbignum`(`make_int`(r)).

8.4. *Si* `sign1` *est faux, changer le signe de r. Rendre* `make_int`(r).

9. *Si x est un petit entier, si $s_1 = 1$, poser $a = x$, sinon $a =$* `make_int`$(-n)$. *Idem pour y et b.*

10. *Si x et y sont des entiers, rendre* `zx`(`ntimes`(x, y)).

11. *Si x et y sont des fractions*

11.1. *Mettre* `is_simplified` *et* `is_not_integer` *à faux.*

11.2. *Si $x = a/b$ et $y = c/d$, si $a = d$, rendre c/b, si $b = c$ rendre a/d (quotient via* `qx`*, comparaison via* `ncmp`*).*

11.3. *Si* `simp_flag` *est faux, rendre* `qx`(ac, bd)*, produit via* `ntimes`.

11.4. *Réduire x et y via* `resimp0`*, recalculer c et d, rendre* `rqstimes1`(x, c, d).

12. *Cas où x ou y est une fraction*

12.1. *Si c'est y, poser $c = a$, $a = y$.*

12.2. *Écrire $x = \pm a/b$, si $b = c$, rendre* `zx`(a).

12.3. *Si* `simp_flag` *est vrai, réduire x via* `resimp0`*, rendre* `rqstimes1`$(x, c, 1)$.

12.4. *Mettre* `is_simplified` *et* `is_not_integer` *à faux, rendre* `qx`(ac, b).

3.5 Division entière

La fonction **rquomod** positionne dans la variable globale `#:ex:mod` le reste de la division, et rend le quotient. Elle est utilisée en interne pour les deux fonctions **quomod** et **modulo**. C'est une fonction générique à deux arguments. Elle n'est cependant pas commutative, ce qui va poser certains problèmes. Par définition, le quotient q et le reste r de la division de a par b satisfont $a = bq + r$, $0 \leq r < |b|$, et q est un entier.

D'après cette définition on voit de suite que a et b doivent être des nombres réels. Nous avons choisi de plus l'option que a et b ne peuvent être des nombres flottants. Ce sont donc des entiers ou des fractions. Notons que si a et b sont positifs, le quotient est la partie entière de a/b, et que si a et b sont entiers, c'est la division usuelle.

Algorithme 20. (Division entière) *Les variables et paramètres utilisés ici sont x et y des entiers, q, r des nombres. Toutes les procédures sont appelées avec x et y comme paramètres. Elles rendent le quotient, et positionnent le reste dans* `#:ex:mod`.

Procédure rqm2. *Cas de deux paramètres positifs x et y.*

1. *Appeler* `push_internal(IS_simp_flag,0)` [ce n'est pas la peine d'extraire des pgcd.]

2. *Calculer q la partie entière de x/y via* `truncate`.

3. *Calculer le reste $r = x - qy$ dans* `mod`.

4. *Appeler* `pop_internal(IS_simpflag)` [dépile la valeur sauvée].
5. *Rendre* q.

Procédure rquomod. *Arguments* x *et* y.

1. *Si* $y < 0$, *rendre* $-\texttt{rquomod}(x, -y)$.
2. *Si* $x < 0$, *soit* $z = \texttt{rquomod}(-x, y)$. *Si le reste est 0, rendre* $-z$, *sinon poser* $m = y - \texttt{mod}$, $z = -z - 1$, $\texttt{mod} = m$ *et rendre* z.
3. *Si* x *ou* y *est rationnel, rendre* $\texttt{rqm2}(x, y)$.
4. *Si* x *est un grand entier, remplacer* x *par* $\texttt{rz\_num}(x)$. *Si* y *est un grand entier, remplacer* y *par* $\texttt{rz\_num}(y)$.
5. *Poser* $x = \texttt{nx}(\texttt{nquomod}(x, y))$, $\texttt{mod} = \texttt{nx}(\texttt{mod})$, *rendre* x.

Fonction quomod. *Argument* x, y.

1. *Erreur si* x *(ou* y*) n'est pas petit entier, un grand entier, ou un rationnel.*
2. *Si* $y = 0$, *erreur: division par zéro.*
3. *Si* $x = 0$, *mettre* 0 *dans* `mod`, *rendre* 0.
4. *Dans les autres cas, rendre* $\texttt{rquomod}(x, y)$.

Fonction modulo. *Argument* x, y.

1. *Appeler* `quomod` *sur* x *et* y.
2. *Rendre* `mod`.

Fonction truncate. *Argument* x [rend la partie entière de x, troncature vers 0].

1. *Si* x *est un entier, rendre* x.
2. *Si* x *est un nombre rationnel, mettre le signe dans* `sign1`, *soit* a *le numérateur,* b *le dénominateur. Rendre* $\texttt{zx}(\texttt{nquomod}(a, b))$.
3. *Si* x *est un flottant, convertir la valeur absolue de* x *en entier, et rajouter le signe.*
4. *Sinon erreur.*

Fonction floor. *Argument* x [rend la partie entière, troncature vers $-\infty$].

1. *Soit* $z = \texttt{truncate}(x)$.
2. *Si* $z > x$ *rendre* $z - 1$ *sinon* z.

Fonction ceiling. *Argument* x [rend la partie entière, troncature vers $+\infty$].

1. *Soit* $z = \texttt{truncate}(x)$.
2. *Si* $z < x$ *rendre* $z + 1$ *sinon* z.

Procédure round1. *Arguments* a *et* b [suppose $b > 0$].

1. *Tester le signe de* a.

1.1. *Si* $a = 0$, *mettre 0 dans le reste, rendre 0.*

1.2. *Si* $a > 0$

1.2.1. *Soient* q *et* r *le quotient et le reste de la division de* a *par* b*, via* `quomod`.

1.2.2. *Soit* $d = b - r$.

1.2.3. *Si* $d < r$*, changer le signe de* d*, incrémenter* q*, sinon poser* $d = r$.

1.2.4. *Mettre* d *dans* `mod`*, et rendre* q.

1.3. *Si* $a < 0$

1.3.1. *Soient* q *et* r *le quotient et le reste de la division de* $-a$ *par* b*, via* `quomod`.

1.3.2. *Soit* $d = b - r$.

1.3.3. *Si* $d \geq r$*, poser* $d = -r$*, sinon* $q = r + 1$.

1.3.4. *Changer le signe de* q.

1.3.5. *Mettre* d *dans* `mod`*, et rendre* q.

1.4. *Sinon erreur.*

Fonction round. *Arguments* x *et* y.

1. *Calculer le signe de* y.

2. *Si* $y > 0$*, rendre* `round1`(x, y).

3. *Si* $y < 0$*, changer le signe de* y*, rendre* $-$`round1`(x, y).

4. *Sinon erreur* [$y = 0$, ou y pas nombre, ou y complexe].

3.6 Fractions continues

En toute généralité, si K est un corps et A une algèbre sur K, x_i une suite d'éléments de A, la fraction continue $[x_0, x_1, \ldots, x_n]$ est définie par

$$[\,] = \infty \qquad [x_0] = x_0$$

$$[x_0, x_1, \ldots, x_m, \ldots] = x_0 + 1/[x_1, \ldots, x_m, \ldots]. \tag{1}$$

Dans le cas où la suite est infinie, la relation de récurrence (1) ne détermine pas vraiment la fraction continue. Une façon de procéder consiste à la définir comme une limite. Pour cela il faut mettre une topologie sur A, et si A n'est pas commutatif faire très attention. Pour simplifier, on suppose que A est un corps de nombres, les réels ou les rationnels, et nous allons souvent faire l'hypothèse que les x_i sont entiers. Comme nous allons le voir, tout réel est représentable en tant que fraction continue, avec x_0 entier, x_i entier positif pour $i > 0$. Réciproquement, dans ce cas la fraction continue existe: dans l'équation (1), tous les termes que l'on inverse sont non nuls. On peut consulter [8] ou [11] pour un certain nombre de propriétés sur les fractions continues, ou les ouvrages originaux de Hurwitz [4, 5, 6].

Si x_i est une suite donnée, posons $p_0 = 0$, $a_0 = 1$, $q_0 = 1$, $b_0 = 0$, et

$$p_{n+1} = a_n, \qquad q_{n+1} = b_n, \qquad a_{n+1} = p_n + a_n x_n, \qquad b_{n+1} = q_n + b_n x_n. \tag{2}$$

La relation (1) donne

$$\frac{p_n + a_n[x_n, x_{n+1}, \ldots]}{q_n + b_n[x_n, x_{n+1}, \ldots]} = \frac{p_{n+1} + a_{n+1}[x_{n+1}, x_{n+2}, \ldots]}{q_{n+1} + b_{n+1}[x_{n+1}, x_{n+2}, \ldots]}, \tag{3}$$

d'où

$$[x_0,\ldots,x_m] = \frac{p_0 + a_0[x_0,x_1,\ldots,x_m]}{q_0 + b_0[x_0,x_1,\ldots,x_m]} = \frac{p_{m+1} + a_{m+1}[\,]}{q_{m+1} + b_{m+1}[\,]} = \frac{a_{m+1}}{b_{m+1}}. \tag{4}$$

Ces relations permettent d'obtenir un algorithme itératif du calcul de la fraction continue finie $[x_0,x_1,\ldots,x_m]$. Notons également que pour tout n on a

$$p_n b_n - a_n q_n = \pm 1,$$

et donc que la fraction a_k/b_k est toujours réduite.

Knuth définit le polynôme $Q_n(x_1,x_2,\ldots,x_n)$ comme valant 1 si $n = 0$, x_1 si $n-1$, et $x_1 Q_{n-1}(x_2,\ldots,x_n) + Q_{n-2}(x_2,\ldots,x_n)$ sinon. La relation (4) est donc $a_{m+1} = Q_{m+1}(x_0,\ldots,x_m)$ et $b_m = Q_m(x_1,\ldots,x_m)$. La relation clé est $Q_n(x_1,x_2,\ldots,x_n) = Q_n(x_n,x_{n-1},\ldots,x_1)$.

Dans le cas d'une suite infinie, on peut dire que $[x_0,x_1,\ldots,x_m,\ldots]$ est la limite des a_i/b_i lorsque i tend vers l'infini, pourvu que cette limite existe.

Soient maintenant a'_i, b'_i, p'_i, q'_i des nombres définis par la même relation de récurrence (2), avec les conditions initiales $p'_1 = 0$, $a'_1 = 1$, $q'_1 = 1$, $b'_1 = 0$. Par récurrence on a

$$p'_n = q_n \qquad a'_n = b_n \qquad p_n = x_0 p'_n + q'_n \qquad a_n = x_0 a'_n + b'_n. \tag{5}$$

On en déduit

$$\frac{a_n}{b_n} = x_0 + 1/\frac{a'_n}{b'_n}.$$

En d'autres termes si $y_i = a_i/b_i$ et $y'_i = a'_i/b'_i$, si une suite converge l'autre converge aussi, et si les limites sont y et y' on a $y = x_0 + 1/y'$. La quantité $[x_0,x_1,\ldots,x_m,\ldots]$ ainsi définie satisfait donc la relation (1).

Par application directe de (4)

$$\frac{a_{n+1}}{b_{n+1}} - \frac{a_n}{b_n} = \frac{a_{n+1}b_n - a_n b_{n+1}}{b_n b_{n+1}} = \frac{\pm 1}{b_n b_{n+1}}.$$

Si les b_i sont positifs et tendent vers l'infini, la suite des a_i/b_i est alternée et converge (de façon précise, la série des différence des a/b est alternée).

Supposons que pour $i > 0$ on ait $x_i \geq 1$. Ceci est le cas si les x_i sont des entiers strictement positifs. La relation $b_{n+1} = b_{n-1} + b_n x_n$ avec les conditions initiales $b_0 = 0$, $b_1 = 1$ montre que la suite des b_i tend vers l'infini au moins aussi vite que la suite de Fibonacci. Il y a donc convergence.

Dans la suite, nous allons nous intéresser uniquement au cas où les x_i sont des entiers, et la suite finie. La valeur de la fraction continue est donc un rationnel. Si x est un nombre réel, non rationnel, l'algorithme « poser $y_0 = x$, x_i partie entière de y_i, $y_{i+1} = 1/(y_i - x_i)$ » permet d'obtenir la fraction continue infinie de x, et elle vérifie $x_i \geq 1$ pour x positif. Il n'y a qu'une seule fraction continue sous ces hypothèses qui s'évalue en x.

On s'intéresse maintenant au cas des fractions continues associés à un rationnel x, et on suppose les x_i entiers (la suite des x_i est donc finie). Il est clair qu'il n'y a pas unicité, par exemple

$$[x_0,x_1,\ldots,x_{n-1},0,x_{n+1},\ldots] = [x_0,x_1,\ldots,x_{n-2},x_{n-1}+x_{n+1},x_{n+2},\ldots]. \tag{6}$$

Il y a trois façons d'obtenir l'unicité :

– représentation classique (ou régulière) : $x_i \geq 1$ pour $i > 0$ et $x_n \geq 2$;

- représentation 01 : pour $i > 0$, les x_i sont 0 ou 1, il n'y a pas deux 0 consécutifs, $x_1 = x_n = x_{n-1} = 1$;
- représentation signée : Pour $i > 0$, $|x_i| \geq 2$, si $x_i = \pm 2$, x_{i+1} a même signe que x_i, et le dernier terme n'est pas -2.

On va montrer l'existence et l'unicité dans chaque cas. En ce qui concerne l'existence, on considère l'algorithme suivant : on écrit $y_0 = x$, puis $y_i = N/D$, $N = Dq + r$, $x_i = q$, $y_{i+1} = D/r$. Par construction, la somme de la fraction continue est x. Dans le cas classique, r est le reste de la division euclidienne, dans le cas 01 on choisit $q = 1$ si $D \leq N$, et dans le cas signé, on choisit $|r| \leq D/2$. La plupart des conditions sont trivialement satisfaites. Pour les autres, il faut travailler un peu.

Cas de la représentation classique

On fait l'hypothèse que $y_i > 1$ pour $i = 0$. Dans le cas où x est un entier, on a $x = [x]$, il n'y a rien à faire. Sinon, la relation $r < D$ entraine $y_{i+1} > 1$. On a de plus $q > y_i - 1$, d'où $q > 0$, et comme q est entier $q \geq 1$. Si par hasard on a $r = 0$, on a $q = y_i > 1$, d'où $q \geq 2$. L'algorithme se terminant si $r = 0$, on a bien $x_n \geq 2$. L'unicité découle du fait que si $i > 0$, on a $[x_i, \ldots, x_n] > 1$, et la partie entière de cet objet est x_i. Remarque : pour comparer $x = [x_0, x_1, \ldots, x_n]$ et $y = [y_0, y_1, \ldots, y_m]$, il suffit de trouver le plus petit i tel que $x_i \neq y_i$, les comparer et décider en faction de la parité de i. Si x est une suite plus courte que y, on peut rajouter $x_{n+1} = +\infty$ pour conclure.

Cas de la représentation 01

Partons d'une représentation 01 et appliquons la relation (6) tant qu'elle est possible pour supprimer des 0. On fait l'hypothèse que tout 0 est suivi d'un 1, $x_1 = x_n = x_{n-1} = 1$. Notons qu'on peut intercaler un 0 entre les deux derniers 1. On appelera représentation 01 modifiée cette autre façon d'écrire. On va montrer qu'elle est unique, caractérisée par les indices i_k tels que $x_{i_k} = x_{i_k+1} = 1$. Par convention $i_0 = 0$, i_1 est le plus petit i positif tel que $x_i = x_{i+1} = 1$. On rajoute n comme dernier indice à cette liste. Pour $k > 0$, on a entre i_{k-1} et i_k une suite alternée de 0 et de 1, de la forme $s_p = 10101\ldots01$, avec p chiffres 1 et $p-1$ chiffres 0. Si on applique (6) en prenant $n + 1 = i_k$, tant que cela est possible, on remplace chaque séquence s_p par le nombre p. On obtient donc $x = [x_0, y_1, \ldots, y_m]$ où $y_k = (i_k - i_{k-1} + 1)/2$. Réciproquement, si $x = [x_0, x_1, \ldots, x_m]$, en remplaçant chaque x_i pour $i > 0$ par la suite s_{x_i}, on obtient une suite de 0 et de 1, en fait une représentation 01 modifiée. Notons que x_m, le dernier terme, est ≥ 2 pour une suite modifiée.

Montrons maintenant que l'algorithme proposé fournit une représentation 01. Si x est donné, on définit x_0 comme la partie entière de x, donc $y_0 > 1$, à moins que x ne soit entier, auquel cas il n'y a rien à montrer. Il est clair que l'algorithme proposé fournit une suite de 0 et de 1, qui commence par 1, sans 0 consécutifs. L'algorithme se termine si $r = 0$, donc si y_n est entier. Soit y_k le dernier non entier dans la liste des y_i. Supposons $x_k = 0$. Alors $y_{k+1} = 1/y_k$, et c'est un entier, donc $x_{k+1} = 1$, $y_{k+2} = 1/(y_{k+1} - 1)$. C'est aussi un entier. Notons que $y_{k+1} \neq 1$ (sinon y_k serait entier) donc y_{k+2} est bien défini, $y_{k+1} = 2$, $x_{k+2} = 1$, c'est la fin de l'algorithme, et on termine avec au moins deux 1. Si maintenant $x_k = 1$, alors $y_{k+1} = 1/(y_k - 1)$ est entier. On a $y_k \neq 1$, donc x_k n'est pas le dernier dans la liste, et tous les x_i suivants sont 1.

Cas de la représentation signée

L'unicité découle de la relation suivante : sauf si le nombre est $[x, 2]$, on a

$$|[x_i, \ldots, x_n] - x_i| < 1/2.$$

En d'autres termes, x_i est l'entier le plus proche de $[x_i, \ldots, x_n]$, sauf si cette quantité est demi entier, auquel cas c'est la partie entière, car $[x_i, -2]$ est exclus. Rappelons que $[x, 2] = x + 1/2$ et $[x, -2] = x - 1/2$.

La relation est triviale pour $i = n$. Supposons-la vraie pour i, montrons-la pour $i - 1$. Par définition, il faut montrer $[x_i, \ldots, x_n] > 2$. Si $x_i \geq 3$, l'hypothèse de récurrence avec inégalité large suffit. Si $x_i = \pm 2$ et $i = n$, la relation est fausse, mais l'inégalité est large. Si $i < n$, on a

$$[x_i, \ldots, x_n] = x_i + 1/[x_{i+1}, \ldots, x_n].$$

Par récurrence, le deuxième terme est du même signe que x_{i+1}, qui a le même signe que x_i par hypothèse. Ceci suffit pour conclure.

L'existence découle de la relation

$$[x_0, 1, x_2, \ldots, x_n] = [x_0 + 1, -x_2 - 1, -x_3, \ldots, -x_n]$$

Cette relation permet de convertir une suite classique vers une représentation signée, et vice-versa. Par exemple, si $x = [1, 1, \ldots, 1, \ldots]$, une première application donne $[2, -2, -1, -1, \ldots]$. Une deuxième application qui chasse le deuxième -1 donne $[2, -2, -2, 2, 1, 1, \ldots]$. On obtient donc un développement périodique de période 4. La bonne façon de faire donne $[2, -3, 3, -3, 3, \ldots]$, un développement périodique de période 2, à partir du second terme.

L'algorithme proposé donne le bon résultat : dans la division $N = Dq + r$, on écrit $|r| < D/2$, avec comme invariant $y_i > 2$. Ceci va donner $|q| > |y_i - 1|$, d'où $q \geq 2$. Notons que q est du même signe que y_i. Dans le cas où r et q ont des signes différents, on aura $|q| > |y_i|$, d'où $|q| \geq 3$. En d'autres termes si $x_i = \pm 2$, x_{i+1} aura le même signe que x_i. Finalement, si y_i est demi-entier, disons $a + 1/2$, il faut prendre $r = \pm 2$, donc $x_i = a$, c'est l'avant dernier terme, et le terme suivant est 2.

Opérations sur les fractions continues

Si $x = [x_0, x_1, \ldots, x_n]$ et $y = [y_0, y_1, \ldots, y_m]$ sont deux fractions continues, il est possible de calculer la somme, le produit, la différence, le quotient, sans avoir à convertir les nombres en rationnels. Dans chaque cas, il suffit de trouver le développement de

$$f(x, y) = \frac{\alpha xy + \beta x + \gamma y + \delta}{\alpha' xy + \beta' x + \gamma' y + \delta'}. \tag{7}$$

L'algorithme utilisé ici est celui implémenté dans Macsyma (MIT AI Laboratory Memo 239, 1972).

On suppose que la représentation est normale, donc $x_0 \leq x < x_0 + 1$, et $y_0 \leq y < y_0 + 1$. L'algorithme consiste à trouver la partie entière C du quotient, puis à remplacer f par $1/(f - C)$. Si la partie entière ne peut être calculée, on ajoute un terme de plus de x ou de y. Si les deux listes se terminent, f devient constant, et on est amené à calculer le développement en fraction continue d'un nombre rationnel.

Si C est la partie entière de $f(x,y)$ alors C est le prochain terme dans le développement cherché, et il faut calculer le développement de

$$\frac{1}{f(x,y)-C} = \frac{\alpha' xy + \beta' x + \gamma' y + \delta'}{(\alpha - C\alpha')xy + (\beta - C\beta')x + (\gamma - C\gamma')y + (\delta - C\delta')}. \tag{8}$$

Si T est le vecteur contenant α, β, γ, δ, α', β', γ' et δ', il suffit, pour i entre 0 et 3, de poser $w = T_{i+4}$, $T_{i+4} = T_i - Cw$, $T_i = w$.

L'algorithme de mise à jour de x est le suivant : si $x = [x_0]$, le numérateur devient $(\gamma + \alpha x_0)y + (\delta + \beta x_0)$. On remplace donc γ et δ par de nouvelles valeurs, et on met α et β à 0. Dans le cas où x n'est pas un entier, on a $x = x_0 + 1/x'$ avec $x' = [x_1, \ldots, x_n]$. Le numérateur devient :

$$\frac{1}{x'}((\alpha x_0 + \gamma)x'y + (\beta x_0 + \delta)x' + \alpha y + \beta). \tag{9}$$

De la même manière, le dénominateur est D/x', les nouvelles valeurs de α, β, γ et δ sont $\alpha x_0 + \gamma$, $\beta x_0 + \delta$, α et β, avec des formules identiques pour le dénominateur.

La partie non triviale est maintenant d'estimer numérateur et dénominateur et de trouver la partie entière du quotient. Supposons $x = x_0 + \epsilon$, $y = y_0 + \eta$, avec $0 \le \epsilon \le 1$ et $0 \le \eta \le 1$. Le numérateur N est alors

$$\alpha\epsilon\eta + (\alpha y_0 + \beta)\epsilon + (\alpha x_0 + \gamma) + \alpha x_0 y_0 + \beta x_0 + \gamma y_0 + \delta$$
$$= A\epsilon\eta + B\epsilon + C\eta + D$$

et $N_1 \le n \le N_2$. Les quantités N_1 et N_2 sont faciles à calculer: par exemple $N_2 = \max(A,0) + \max(B,0) + \max(C,0) + D$. On encadre de même le dénominateur, ce qui donne un encadrement de f.

La stratégie adoptée ici est de calculer $f_0 = f(x_0, y_0)$, $f_1 = f(x_0, y_0 + 1)$, $f_2 = f(x_0 + 1, y_0)$ et $f_3 = f(x_0 + 1, y_0 + 1)$. On calcule d'abord A, B, C et D, et le numérateur de f_2 est par exemple $B + D$. Dans le cas où f n'a pas de singularité sur le carré $0 \le \epsilon \le 1, 0 \le \eta \le 1$, alors le maximum de f est obtenu sur le carré, donc est le plus grand des f_i. Il en est de même du minimum. Ceci est facile à montrer: à x ou y fixés, f est une homographie, donc est monotone. Dans le cas où f a une singularité, les dénominateurs des divers f_i n'ont pas tous le même signe. Dans ce cas, on ne peut rien conclure. Admettons que les dénominateurs ont le même signe. Si tous les f_i ont la même partie entière, cette partie entière est la valeur C cherchée.

Dans le cas où les dénominateurs n'ont pas le même signe, ou si un des dénominateurs s'annule, il faut prendre un terme de plus de x ou y. Soit $\delta_x = f_2 - f_0$, $\delta_y = f_1 - f_0$. Dans le cas où l'une des différences n'est pas définie, on prend ∞ à la place. En fait, on utilise la partie entière de f_i et non f_i pour ce calcul; ceci rend le code un peu plus efficace. Si $\delta_x = 0$, c'est que f dépend très peu de x, on prend donc un terme de plus de y. Si c'est δ_y qui est nul, on avance dans x. Notons que si x est épuisé, alors f ne dépend plus de x et $\delta_x = 0$ (sauf si cette quantité est indéfinie, mais si de plus y est vide, f est constant, et si $f = \infty$, c'est la fin de l'algorithme). Si aucun δ_x, δ_y n'est nul, on choisit le plus grand. Si c'est δ_x, on avance dans x. Si tous les deux sont infinis, on choisit x par exemple.

Racines carrées

Les formules qui suivent sont utilisées par l'algorithme de factorisation de Morrison et Brillhart. Nous allons les utiliser pour montrer que toute solution d'une équation de degré 2 à coefficients entiers possède un développement en fraction continue périodique, et réciproquement.

Supposons d'abord le développement périodique, $x_{m+k} = x_k$ pour tout $k \geq 0$. Par application de (4) on a

$$x = \frac{p_m + a_m x}{q_m + b_m x}$$

donc $b_m x^2 + (q_m - a_m)x - p_m = 0$ et

$$x = \frac{a_m - q_m \pm \sqrt{(q_m - a_m)^2 + 4p_m b_m}}{2b_m}.$$

Dans le cas où les x_i sont positifs, $p_m b_m > 0$, la racine carrée est réelle, et il faut prendre la racine positive. Omettons l'indice m dans la suite. La racine est aussi $\sqrt{(q+a)^2 \pm 4}$. Cette quantité n'est pas un entier : si $n^2 \pm 4 = m^2$ où n et m sont des entiers, alors cette équation modulo 2 montre que n et m ont la même parité. Quitte à échanger n et m, on peut supposer $n^2 = m^2 + 4$. On peut supposer $n \geq 0$ et $m \geq 0$. Alors $m < n < m + 2$, absurde, sauf si $m = 0$ et $n = 2$. Comme $q + a > 0$, le seul cas où la racine peut être un entier est $q + a = 2$, et la racine est $\sqrt{(q+a)^2 - 4}$, i.e., période de longueur paire. Si la période est de longueur au moins 2, alors $a + q = 2 + x_0 x_1 > 2$, absurde.

Si maintenant x est périodique à partir du rang k, on a $x = (p+ay)/(q+by)$, avec $Ay^2+By+C = 0$. Écrivons $y = (\alpha x + \beta)/(\gamma x + \delta)$. Alors

$$A(\alpha x + \beta)^2 + B(\alpha x + \beta)(\gamma x + \delta) + C(\gamma x + \delta)^2 = 0.$$

En développant, on obtient une équation du second degré. Le coefficient dominant est $A\alpha^2 + B\alpha\gamma + C\gamma^2$. Il est non nul, car l'équation qui donne y n'a pas de solutions rationnelles.

Réciproquement, toute solution d'une équation du second degré est de la forme $x = (a \pm \sqrt{b})/c$. On suppose que b n'est pas un carré parfait. On va montrer que cet objet a un développement en fraction continue périodique, et donner un algorithme de calcul. Quitte à changer les signes de a et c, on peut supposer que la racine est positive. On va de plus supposer que c divise $a^2 - b$. C'est une condition importante : elle va permettre de borner les quantités qui apparaissent dans la suite. Si on multiplie a et c par d, et b par d^2, l'expression x reste la même, mais $(a^2 - b)/c$ est multipliée par d. Si c ne divise pas $a^2 - b$, il suffit de choisir $d = c$.

Notons que pour le calcul de $\sqrt{n}$, où n est entier, on a $c = 1$. Pour calculer $\sqrt{u/v}$, où u et v sont premiers entre eux, il faut (et suffit) d'écrire $x = \sqrt{uv}/v$. Par exemple, si $x = \sqrt{2/3}$, les y_i obtenus sont $\sqrt{6}/3$, $\sqrt{6}/2$, $2 + \sqrt{6}$, $(2 + \sqrt{6})/2$, etc. Ils sont tous de la forme $(a + \sqrt{6})/c$. Le développement est $[0, 1, 4, 2, 4, 2, \ldots]$.

Le but du jeu dans l'algorithme de Morrison et Brillhart est de trouver des nombres p_i, q_i et V_i tels que

$$p_i^2 - nq_i^2 = V_i. \qquad (*)$$

À titre de curiosité, l'un des V_i obtenus est ± 1, et $p_i - q_i\sqrt{n}$ est un élément inversible de l'anneau $Z[\sqrt{n}]$. La procédure `Qunit` calcule un tel objet. Supposons que p_i et q_i sont très grands, et V_i est petit. La relation $(*)$ montre que p_i/q_i approche très bien $\sqrt{n}$. C'est la réciproque qui va servir : si p_i/q_i est l'un des approximants de $\sqrt{n}$ alors V_i est petit, au plus $2\sqrt{n}$, et la relation $(*)$ est non triviale. De façon précise,

$$p_i^2 = V_i \pmod{n} \qquad (**)$$

est non triviale. De plus, si p est un nombre premier divisant V_i, alors n est un carré modulo p. Le principe de l'algorithme est le suivant : on choisit tous les premiers r_i satisfaisant la condition

précédente et on écrit

$$V_i = \prod r_j^{\alpha_{ij}}.$$

Ceci donne un certain nombre de vecteurs α_i. On prend des combinaisons linéaires à coefficients entiers de ces vecteurs de telle sorte que toutes les composantes deviennent paires, en procédant comme suit : on itère sur j. Si tous les α_{i1} sont pairs, il n'y a rien à faire. Sinon, on prend un vecteur α_i avec α_{i1} impair, on l'ajoute à tous les autres, et on jette ce vecteur. On continue avec $j = 2$, $j = 3$, etc. Les composantes déjà traitées restent paires, car la somme de deux nombres pairs est paire. Finalement, on obtient des relations de type $(**)$ pour lesquelles V_i est un carré parfait, donc des relations du type $(p-V)(p+V) = 0 \pmod n$. Avec de la chance, cette relation est non triviale, et le pgcd de $p-V$ et n est un facteur non trivial de n. L'implémentation effective de l'algorithme est plus subtile, elle sera détaillée dans la suite.

On va maintenant donner l'algorithme du calcul du développement en fractions continues de $x = (\sqrt{n} - U)/V$. La preuve est inspirée de Knuth [8, exercice 4.5.3.12], les notations sont un peu différentes. On cherche des entiers a_i, des nombres x_i, tels que

$$x_i = a_i + 1/x_{i+1},$$

et a_i est la partie entière de x_i. On a $x_0 = x$. Comme le suggère l'exemple de $\sqrt{2/3}$, on va supposer

$$x_i = \frac{\alpha_i + \sqrt{n}}{\beta_i}. \tag{10}$$

Posons $\gamma_{-2} = 0$, $\gamma_{-1} = 1$, $\delta_{-1} = 0$, $\delta_0 = 1$, et

$$\gamma_{i+1} = \gamma_{i-1} + a_{i+1}\gamma_i. \tag{11}$$

$$\delta_{i+1} = \delta_{i-1} + a_{i+1}\delta_i. \tag{12}$$

Ce sont les relations (2), donc, pour tout X, on a

$$[a_0, a_1, \ldots, a_i, X] = \frac{\gamma_i X + \gamma_{i-1}}{\delta_i X + \delta_{i-1}}. \tag{13}$$

Admettons

$$\frac{1}{x_{i+1}} = \frac{\sqrt{n} - \alpha_{i+1}}{\beta_i}, \qquad x_{i+1} = \frac{\sqrt{n} + \alpha_{i+1}}{\beta_{i+1}}. \tag{14}$$

Ceci nous donne la contrainte

$$n - \alpha_{i+1}^2 = \beta_i \beta_{i+1}. \tag{15.1}$$

La relation $x_i = a_i + 1/x_{i+1}$ donne

$$\alpha_i = \beta_i a_i - \alpha_{i+1}. \tag{16}$$

La dernière relation est

$$a_i = \left\lfloor \frac{\sqrt{n} + \alpha_i}{\beta_i} \right\rfloor. \tag{17}$$

On notera g la partie entière de n. Dans le cas $\beta_i > 0$, a_i est le quotient de $g + \alpha_i$ par β_i, dans le cas contraire c'est le quotient de $-g - \alpha_i - 1$ par $-\beta_i$. L'algorithme utilise donc uniquement n et g. Il n'est nul besoin de connaitre autre chose sur $\sqrt{n}$ que g.

Analysons la relation (15.1). Si on fait la différence entre le rang i et le rang $i+1$, on obtient $\alpha_i^2 - \alpha_{i+1}^2 = \beta_i(\beta_{i+1} - \beta_{i-1})$. Le membre de gauche est $(\alpha_i - \alpha_{i+1})(\alpha_i + \alpha_{i+1}) = (\alpha_i - \alpha_{i+1})\beta_i a_i$, par utilisation de (16) d'où

$$(\alpha_i - \alpha_{i+1})a_i = \beta_{i+1} - \beta_{i-1}. \tag{15.2}$$

On en déduit que si β_0 et β_1 sont des entiers, alors tous les β_i sont entiers.

Les conditions initiales sont $\beta_0 = V$, $\alpha_0 = -U$, $\alpha_1 = U + a_0 V$, et $\beta_1 = (n - \alpha_1^2)/V$. La relation clé est donc que V divise $n - \alpha_1^2$. Or ceci est $n - U^2$ modulo V. On se sert donc de l'hypothèse que V divise $n - U^2$ pour conclure que l'algorithme calcule le développement en fraction continue de x. (pour passer du rang i au rang $i+1$, on utilise, dans l'ordre, les relations 17, 16, et 15.2)

Posons $y = (-\sqrt{n} - U)/V$, $1/y_{i+1} = (-\sqrt{n} - \alpha_{i+1})/\beta_i$. Les relations (14), (15) et (16) montrent que

$$x = [a_0, a_1, \ldots, a_i, x_{i+1}]. \tag{18.1}$$

Si on remplace $\sqrt{n}$ par $-\sqrt{n}$, on obtient

$$y = [a_0, a_1, \ldots, a_i, y_{i+1}]. \tag{18.2}$$

Cette dernière relation n'est pas le développement en fraction continue de y. Il faudrait pour cela que $y_i > 0$; nous allons dans un instant montrer le contraire. Appliquons (13):

$$x = \frac{\gamma_i x_{i+1} + \gamma_{i-1}}{\delta_i x_{i+1} + \delta_{i-1}}$$

et inversons. On en déduit:

$$\frac{1}{x_{i+1}} = -\frac{\delta_i}{\delta_{i-1}} \frac{x - \gamma_i/\delta_i}{x - \gamma_{i-1}/\delta_{i-1}}, \qquad \frac{1}{y_{i+1}} = -\frac{\delta_i}{\delta_{i-1}} \frac{y - \gamma_i/\delta_i}{y - \gamma_{i-1}/\delta_{i-1}}. \tag{19}$$

Première utilisation de ces formules: on les multiplie entre elles. On obtient

$$\frac{-\beta_{i+1}}{\beta_i} = \frac{\alpha_{i+1}^2 - n}{\beta_i^2} = \frac{1}{x_{i+1} y_{i+1}} = \frac{(\delta_i x - \gamma_i)(\delta_i y - \gamma_i)}{(\delta_{i-1} x - \gamma_{i-1})(\delta_{i-1} y - \gamma_{i-1})}.$$

On en déduit qu'il existe une constante C, indépendante de i telle que

$$(\delta_i x - \gamma_i)(\delta_i y - \gamma_i) = (-1)^i \beta_{i+1}.C$$

Si on écrit x et y en fonction de U et V, et qu'on évalue la constante en $i = 0$ on obtient

$$\frac{U^2 - n}{V}\delta_i^2 + 2U\delta_i\gamma_i + \gamma_i^2 = (-1)^{i+1}\beta_{i+1}. \tag{20}$$

Dans le cas de $x = \sqrt{n}$ on a donc

$$-n\delta_i^2 + \gamma_i^2 = (-1)^{i+1}\beta_{i+1}. \tag{20.1}$$

C'est la relation voulue pour l'algorithme de Morrison et Brillhart.

Posons maintenant $r_i = g - \alpha_{2i-1}$, $s_i = g - \alpha_{2i}$, $q_i = a_{2i}$, $p_i = a_{2i+1}$, $P_i = \beta_{2i}$, $Q_i = \beta_{2i-1}$, $A_i = \gamma_{2i-2}$ et $B_i = \gamma_{2i-1}$. La relation (11) s'écrit

$$A_{i+1} = A_i + q_i B_i \tag{21.1}$$

$$B_{i+1} = B_i + p_i A_{i+1}. \tag{21.2}$$

La relation (15.2) est

$$Q_{i+1} = Q_i + q_i(r_{i+1} - s_i) \tag{22.1}$$

$$P_{i+1} = P_i + p_i(s_{i+1} - r_{i+1}). \tag{22.2}$$

La relation (16) est

$$2g - s_i = P_i q_i + r_{i+1} \tag{23.1}$$

$$2g - r_{i+1} = Q_{i+1} p_i + s_{i+1}. \tag{23.2}$$

et la relation (20) est

$$A_i^2 + Q_i = n\delta_{2i-2}^2 \tag{24.1}$$

$$B_i^2 - P_i = n\delta_{2i-1}^2. \tag{24.2}$$

La seconde relation (19) montre que la suite est périodique à partir d'un certain rang. En effet, la quantité γ_i/δ_i tend vers x par construction, et le deuxième quotient tend vers 1, car $x \neq y$. Comme le facteur δ_i/δ_{i-1} est positif, on en déduit que si i est assez grand, on a $y_i < 0$. Cette condition s'écrit $(\sqrt{n} - \alpha_i)/\beta_i > 0$ et $\beta_i/(\sqrt{n} + \alpha_{i+1}) > 0$.

Dénotons par $P(x_i)$ la propriété $0 < \sqrt{n} - \alpha_i < \beta_i$ et par $Q(x_i)$ la propriété $0 < \beta_i < \sqrt{n} + \alpha_{i+1}$. Ces conditions sont un peu plus fortes, car elles entraînent que les quantités précédentes sont entre 0 et 1, et non plus simplement positives. Pour $i > 0$ on a $x_i > 0$, donc si i est assez grand on a $\beta_i > 0$.

Faisons maintenant l'hypothèse suivante: $\beta_i > 0$, $\beta_{i-1} > 0$. Comme $0 < 1/x_{i+1} < 1$, la relation (14) montre que $\alpha_{i+1} < \sqrt{n}$ et $\alpha_{i+1} > \sqrt{n} - \beta_i$. On a également $\alpha_i < \sqrt{n}$ et $\alpha_i > \sqrt{n} - \beta_{i-1}$. Si $\beta_i < \sqrt{n}$, on en déduit $\alpha_{i+1} > 0$. Sinon, par (16), $\alpha_{i+1} = \beta_i a_i - \alpha_i \geq \sqrt{n} - \alpha_i$, d'où $\alpha_{i+1} > 0$ (rappel $a_i \geq 1$). En particulier $0 < \alpha_{i+1} \leq g$. Finalement (15.1) montre que $\beta_{i+1} > 0$ et la relation (16) donne $\beta_i < \sqrt{n} + \alpha_{i+1}$, d'où $0 \leq \beta_i \leq 2g$.

On vient donc de montrer que « $\beta_i > 0$ et $\beta_{i-1} > 0$ » entraîne $0 < \alpha_{i+1} \leq g$, $0 < \beta_i \leq 2g$, et donc la relation au cran suivant. Ces relations sont donc vraies pour tout $i' \geq i$. La suite des α_i et des β_i est bornée, donc périodique, Par ailleurs, si on est dans la période la condition $Q(x_i)$ est vraie. Une des conséquences de (15.1) est

$$\frac{\sqrt{n} - \alpha_{i+1}}{\beta_{i+1}} = \frac{\beta_i}{\sqrt{n} + \alpha_{i+1}}.$$

En d'autres termes, si $Q(x_i)$ est vraie et $\beta_{i+1} > 0$, alors $P(x_{i+1})$ est vraie. Notons que la condition « $Q(x_i)$ et $\beta_{i+1} > 0$ » se propage (elle entraîne $\beta > 0$).

Soit maintenant $g(x_{i+1})$ la fonction définie comme suit:

$$\beta = \frac{n - \alpha_{i+1}^2}{\beta_{i+1}} \qquad a = \left\lfloor \frac{\alpha_{i+1} + \sqrt{n}}{\beta} \right\rfloor \qquad \alpha = \beta a - \alpha_{i+1}$$

et $g = (\alpha + \sqrt{n})/\beta$. Par (15.1) on a $\beta = \beta_i$, et par (16) $(\alpha_{i+1} + \sqrt{n})/\beta_i = a_i + (\sqrt{n} - \alpha_i)/\beta_i$. La condition $P(x_i)$ nous dit alors que $a = a_i$, donc $g(x_{i+1}) = x_i$.

Comme la suite est périodique, il existe un k_0 minimal à partir duquel on a $x_{i+m} = x_i$, pour tout i. Si $i \geq k_0$ alors $P(x_i)$ est vrai: en effet, il existe j aussi grand qu'on veut tel que $x_i = x_j$. Si j est assez grand alors $Q(x_{j-1})$ est vrai, donc $P(x_j)$ aussi. Réciproque. Montrons

d'abord que $P(x_i)$ est faux pour $i = k_0$. En fait, si la relation était vraie, on aurait $x_i = g(x_{i+1})$. Or $x_{i+m} = g(x_{i+m+1}) = g(x_i)$ car P est vraie pour x_{i+m}. Ceci contredit la minimalité de k_0. Finalement, prenons le plus petit i pour lequel soit $P(x_i)$, $Q(x_i)$ et $\beta_{i+1} > 0$ sont vrais, soit $P(x_i)$ et $Q(x_{i-1})$ sont vrais. Dans les deux cas, il existe $j \leq i$ tels que $Q(x_j)$ est vrai, $\beta_j > 0$ et $\beta_{j+1} > 0$. Comme cela a été vu plus haut, si ceci est vrai pour un j_0, cela reste vrai pour tout $j \geq j_0$. On a donc $P(x_j)$ vrai pour tout $j > j_0$, donc vrai pour tout $j \geq i$. On en déduit $i \geq k_0$. Comme i est pris le plus petit possible c'est que $i = k_0$.

Application: considérons le cas de $\sqrt{u/v}$ (u et v premiers entre eux), et notons par h la partie entière de $\sqrt{u/v}$. La condition $Q(x_0)$ est équivalente à $u > v$. La condition $P(x_0)$ s'écrit $u < v$. La suite n'est donc pas périodique à partir du début. Supposons $u > v$ (sinon le premier terme du développement est 0). On a $x_1 = (\sqrt{uv} + hv)/(u - vh^2)$. La condition $P(x_1)$ est $\sqrt{u/v} + h > 0$. Elle est donc vraie. Dans ce cas la suite est périodique à partir du second terme. Dans le cas $u < v$, elle est périodique à partir du troisième (ceci se constate sur l'exemple de $\sqrt{2/3}$). Supposons à nouveau $u > v$. On a $g(x_1) = x_0 + h$. en d'autres termes, la suite $x_0 + h$ est périodique à partir du premier terme.

Dans le cas particulier de $\sqrt{n}$, on a $v = 1$, donc $u > v$ et $h = g$. De plus, les relations $\beta \geq 1$ et $\alpha \leq g$ montrent que $a_k = 2g$ n'est possible que si $\beta = 1$ et $\alpha = g$. En d'autres termes, $2g$ n'apparait qu'une seule fois dans la période. Donc: si k est la premier indice pour lequel $a_k = 2g$, alors k est la plus petite période de la suite a_i.

On peut se poser la question dans le cas général: si $u > v$, m est la plus petite période de la suite des x_i, M la période de la suite des a_i, on sait $a_m = 2h$, donc $a_M = 2h$. Cependant rien ne garantit que M est le plus petit indice pour lequel cette relation est vraie.

Dans le calcul de $\sqrt{n}$, par les relations (20) à (24), les choses sont un peu plus simples: on suppose P_i et Q_{i+1} positifs. La relation $x_{i+1} > 1$ dit $0 \leq g - \alpha_{i+1} < \beta_i$. En d'autres termes, dans (23.1) et (23.2), on a $0 \leq r_{i+1} < P_i$ et $0 \leq s_{i+1} < Q_{i+1}$. On obtient donc q_i et r_{i+1} par division euclidienne dans (23.1), p_i et s_{i+1} par division dans (23.2). Ces relations sont utilisées par l'algorithme de Morrison et Brillhart.

Algorithme 21. (Fonctions de base sur les fractions continues) *Sauf mention explicite du contraire, tous les arguments et variables locales sont des entiers, L est une liste. Pour construire la liste $(x_0, x_1, \ldots)$, on insère chaque x_i dans la liste via* `cons`*, ceci donne le résultat à l'envers, qui est remis à l'endroit via* `nreverse`.

Fonction cf-to-rational1. *Argument L.* [Si $L = (a_0, \ldots, a_n)$, rend $[a_0, \ldots, a_n]$, calculé par (1)]

1. *Si le premier élément de L est la chaîne* `"-"`*, mettre s à -1, avancer dans L, sinon mettre s à 1.*
2. *Renverser la liste L. Soit x le premier élément de L. Avancer dans L.*
3. *Tant que L est non vide, soit y l'élément suivant. Remplacer x par $y + 1/x$.*
4. *Rendre sx.*

Fonction cf-to-rational. *Argument L.* [Si $L = (a_0, \ldots, a_n)$, rend $[a_0, \ldots, a_n]$, calculé par (4)]

1. *Si le premier élément de L est la chaîne* `"-"`*, mettre s à -1, avancer dans L, sinon mettre s à 1.*
2. *Poser $p_1 = 0$, $p_2 = 1$, $q_1 = 1$, $q_2 = 0$.*

3. *Pour tout élément x de la liste L, poser $p' = p_2$, $q' = q_2$, $p_2 = p_1 + p_2 x$, $q_2 = q_1 + q_2 x$, $p_1 = p'$, $q_1 = q'$.*

4. *Rendre sp_2/q_2.*

Procédure ecrifc0. *Arguments n, z, p.* [Rend le développement en fraction continue de n avec p termes, ou l'imprime. Si $z = 1$, $z = 0$ ou $z = -1$, on utilise la représentation classique, 01, ou signée]

1. *Si $p < 0$, remplacer p par $-1 - p$, positionner C à vrai, L à la liste vide.* [p est le nombre de termes, si C est vrai, on rend une liste d'objets L, sinon on les imprime.]

2. *Si $n < 0$, $z \neq 1$, remplacer n par $-n$. Si C est vrai, ajouter un signe '-' au résultat partiel, sinon l'imprimer.*

3. *Soient x et y les numérateur et dénominateur de n. Si C est faux, imprimer une barre de fraction.*

4. *Tant que p est positif, et s est faux.*

4.1. *Écrire $x = qy + r$, division euclidienne si $z = 1$, r le plus petit possible en valeur absolue si $z = -1$, et sinon, $q = 0$ ou $q = 1$.*

4.2. *Imprimer q, ou le mettre dans le résultat partiel.*

4.3. *Poser $x = y$, $y = r$.*

4.4. *Si $y = 0$, fin de la boucle.*

4.5. *Si C est faux, et z non nul imprimer un espace.*

4.6. *Décrémenter p.*

5. *Si y est non nul, ajouter x/y au résultat partiel, ou imprimer un espace, x, un trait de fraction, puis y.*

6. *Si C est vrai, rendre le résultat L, sinon imprimer un trait de fraction.*

Procédure cfeval. *Argument x.*

1. *Si x est un entier, rendre la liste contenant x seul.*

2. *Si x est une liste non vide ne contenant que des entiers, rendre x.*

3. *Si x est un nombre rationnel, utiliser* `ecifc0` *avec les bons paramètres pour le convertir en fraction continue.*

4. *Sinon erreur.*

Fonction cfop. *Fonction à deux ou trois arguments f, x, y.*

1. *Erreur si f n'est pas $+$, $-$, $*$ ou $/$.*

2. *Erreur si $+$ ou $*$ ne sont pas appelés avec 2 arguments.*

3. *Si $-$ est appelé avec 1 argument y, poser $x = 0$. Si $/$ est appelé avec un argument y, poser $x = 1$.*

4. *Évaluer x et y via* `cfeval`*. Initialiser L à la liste vide.*

5. *Pour la somme, utiliser $T = [0, 1, 1, 0, 0, 0, 0, 1]$, pour $-$, $T = [0, 1, -1, 0, 0, 0, 0, 1]$, pour $*$ utiliser $T = [1, 0, 0, 0, 0, 0, 0, 1]$ et pour $/$ utiliser $T = [0, 1, 0, 0, 0, 1, 0, 0]$.* [les 8 éléments de T seront notés α, β, γ, δ, α', β', γ' et δ'.]

6. *Tant que au moins un des 4 derniers éléments de T est non nul*

6.1. *Soit A le premier élément de x, B le premier élément de y.*

6.2. *Calculer* `cfeval_all`(T, a, b, P)*, où P est un vecteur de longueur 4.*

6.3. *Si le résultat est vrai, soit $C = P_0$:*

6.3.1. *Ajouter C au résultat partiel.*

6.3.2. *Poser $w = \alpha'$, $\alpha' = \alpha - Cw$, $\alpha = w$.*

6.3.3. *Répéter 3 fois, avec β, β', puis γ, γ' et δ, δ'.*

6.3.4. *Continuer la boucle.*

6.4. *Poser $d_x = P_2 - P_0$, $d_y = P_1 - P_0$* [si un argument est faux, la différence est faux]. *Si d_y est 0, mettre à jour x, si d_x est 0, mettre à jour y. Si d_x et d_y sont entiers et $|d_x| < |d_y|$, mettre à jour y sinon x.*

6.5. [Mise à jour de x.] *Si A est le dernier élément de x : Poser $\gamma = A\alpha + \gamma$, $\delta = A\beta + \delta$, $\alpha = 0$, $\beta = 0$, idem avec des primes.*

6.6. [Mise à jour de x, il y a des termes après A.] *Poser $u = \alpha$, $v = \beta$. Poser $\alpha = A\alpha + \gamma$, $\beta = A\beta + \delta$, $\gamma = u$, $\delta = v$. Idem avec des primes. Avancer dans x.*

6.7. [Mise à jour de y] *Faire la même chose en utilisant B au lieu de A, y au lieu de x, et échanger les rôles de β et γ.*

7. *Rendre le résultat L.*

Procédure cfeval_all. *Argument T, a, b, et un vecteur P.*

1. *T contient α, β, γ, δ, et 4 autres éléments notés avec des primes.*

2. *Calculer $A = \alpha$, $B = \alpha b + \beta$, $C = \alpha a + \gamma$, et $D = \alpha ab + a\beta + b\gamma + \delta$.*

3. *Poser $n_0 = D$, $n_1 = D + C$, $n_2 = D + B$, $n_3 = A + B + C + D$.*

4. *Calculer de même les d_i, en utilisant α' etc. au lieu de α, etc.*

5. *Si d_i est non nul, remplacer n_i par la partie entière de n_i/d_i, sinon par faux.*

6. *Mettre les n_i dans le vecteur P.*

7. *Si l'un des d_i est nul, ou les d_i n'ont pas tous le même signe, rendre faux.*

8. *Si les n_i ne sont pas égaux, rendre faux.*

9. *Sinon rendre vrai.*

Procédure cfisqrt. *Paramètre n, a_0.*

1. *Poser $L = [a_0]$, $q_2 = 1$, $m_1 = a_0$, $q_1 = n - a_0^2$.*

2. *Poser $N = (m_1 + a_0)/q_1$ (quotient entier). Ajouter N à L. Fin si $N = 2a_0$, rendre L.*

3. *Poser $m = Nq_1 - m_1$, $q = q_2 + N(m_1 - m)$.*

4. *Poser $q_2 = q_1$, $q_1 = q$, $m_1 = m$, aller en* **2**.

Fonction CFsqrt. *Argument x.*

1. *Erreur si x n'est pas un entier positif.*

2. *Soit a_0 la partie entière de la racine carrée de x.*

3. *Si x est un carré parfait, rendre $[a_0]$.*

4. *Rendre* `cfisqrt`(x, a_0).

Fonction Qunit. *Argument x.*

1. *Erreur si x n'est pas un entier positif.*
2. *Soit a_0 la partie entière de la racine carrée de x.*
3. *Si x est un carré parfait, erreur.*
4. *Appeler* `cfisqrt` *sur x et a_0.*
5. *Supprimer le dernier terme de la liste, convertir la liste en fraction a/b.*
6. *Rendre $a + b\sqrt{n}$.*

Chapitre 4

Arithmétique générique et complexe

Un nombre complexe est formé de deux champs, une partie réelle et une partie imaginaire, ce sont deux nombres réels. Contrairement à l'option choisie par Common Lisp, les parties réelles et imaginaires peuvent être de type différent. Pour l'instant, ce sont des rationnels ou des flottants, on espère un jour autoriser des BigFloat, mais cela pose des problèmes de conversions, comme nous allons le voir dans la suite.

4.1 Types et conversions

La fonction **integerp** rend vrai si son argument est un entier ou un chiffre. Rappelons qu'une fraction n'est jamais un entier, donc que contrairement à la documentation Lisp, la fonction **integerp** appelée sur une fraction ne la réduit pas. La fonction **rationalp** rend vrai si son argument est un rationnel ou un entier (ou un chiffre). La fonction **realp** rend vrai si son argument est un réel (un flottant, un BigFloat, ou un objet pour lequel **rationalp** rend vrai). La fonction **complexp** rend vrai si son argument est un nombre complexe (ceci inclut les nombres réels, donc n'est pas consistant avec la définition de Common Lisp). La fonction **truecomplexp** rend vrai si son argument est un nombre dont la représentation interne est celle d'un nombre complexe (ceci exclut les nombres réels).

Les fonctions **realpart** et **imagpart** rendent la partie réelle et imaginaire d'un nombre complexe. Si ce nombre est réel, la première rend le nombre, la seconde rend 0 (ou 0.0 si le nombre est flottant).

La fonction **float** convertit son argument en flottant. Elle n'est pas définie pour les complexes, et peut provoquer des débordements si son argument est trop grand. Si l'argument est une fraction, elle est réduite au préalable (pour éviter dans certains des dépassements de capacité).

La fonction **fix** ou **floor** rend la partie entière (au sens usuel) d'un nombre. Elle n'est définie que pour les nombres réels. Il peut y avoir des problèmes de précision pour les nombres flottants. La fonction **ceiling** convertit un nombre en entier en tronquant dans l'autre sens, la fonction **truncate** convertit son argument en tronquant en direction de 0. La fonction **round** est définie

comme en Lisp. Sa sémantique est peu claire, elle prend deux arguments. Pour plus de détails, on peut consulter [10, p. 216], en remarquant toutefois que nos fonctions ne rendent qu'une valeur, et que même si la variable `#:ex:mod` contient parfois le reste correct, ce n'est qu'un heureux hasard.

4.2 Fonctions trigonométriques

La fonction **phase** appliquée à un nombre complexe c rend sa phase. Si le module (valeur absolue) est r, la phase ϕ est telle que $c = re^{i\phi}$. Cette phase est entre $-\pi$ et π, elle est calculée en utilisant **atan2** qui est une fonction à deux arguments x et y qui calcule **atan** (l'arc tangente) du quotient x/y, mais utilise les signes de façon conventionnelle (voir par exemple [10, p. 208]) pour avoir une détermination du résultat entre $-\pi$ et π. La fonction **conjugate** rend le conjugué de son argument si c'est un complexe, son argument sinon.

Les fonctions **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, leur équivalent hyperbolique, de même que les fonctions **log**, **log10**, **sqrt**, **power** et **exp** existent. Elles font leurs calculs en complexe ou flottant. La méthode de calcul n'est pas optimale. Pour l'instant il n'y a pas moyen de faire des calculs en BigFloat (par exemple, si on appelle **asin** sur l'entier 1, le résultat est $\pi/2$, et l'arithmétique générique n'est pas appelée).

Notons que la fonction ****** prend deux arguments x et n et calcule x^n. Dans le cas où n n'est pas entier, ceci se fait en appelant la fonction **power**, mais sinon elle utilise des multiplications.

Notons également que la fonction **powermod** prend trois arguments a, b et c. Elle calcule a^b modulo c où $c > 0$ et $b \geq 0$ de façon efficace. L'inverse de a modulo c peut être calculée via la fonction **bezout**.

4.3 Autres fonctions

La fonction **fib** calcule un nombre de Fibonacci F_n, où F_n, défini par $F_1 = F_2 = 1$ et $F_n = F_{n-1}+F_{n-2}$, vérifie la propriété $F_{2n} = F_n(2F_{n+1}-F_n)$, $F_{2n+1} = F_{n+1}^2+F_n^2$ et $F_{2n+2} = F_{n+1}(2F_n+F_{n+1})$. Pour $n \leq 22$ on utilise la définition itérative, et sinon, les trois formules précédentes. Pour être plus efficace, les fonctions génériques sont remplacées par des appels à `ntimes` et `nadd`. A titre d'exemple, le calcul de F_{10000} prend 0.15 secondes sur une Sparc Station 2, le résultat est un nombre avec environ 2000 chiffres, et le nombre s'imprime en 0.32 secondes.

La fonction *(***fact** *n)* calcule la factorielle de n pour l'entier n via $fact(1, 1, n)$. La fonction $fact(g, i, d)$ calcule le produit $g(g+i)\dots(g+\alpha i)$ où α est le plus grand entier tel que $g + \alpha i \leq d$. Si le nombre de termes dans le produit est au plus 10 (i.e. $d - g < 10i$), on fait les produits, sinon on calcule le produit de $fact(g, 2i, d)$ et $fact(g+i, 2i, d)$. À titre d'exemple, la factorielle de 10000 se calcule en 35 secondes, le résultat a de l'ordre de 35 000 chiffres, le temps d'impression est de 34 secondes. A titre de comparaison, le logiciel de calcul formel Maple (qui possède une arithmétique en base 10) calcule ce nombre en 230 secondes, et l'imprime en 20 secondes (bien entendu, comme la base est 10, l'algorithme d'impression est trivial, il faut cependant du temps pour imprimer les 35 000 chiffres, en moyenne une seconde par page d'écran).

Les fonctions **gcd** et **bezout** calculent le pgcd et la relation de Bezout entre deux entiers. Nous avons expliqué le code interne `npgcd` et `nbezout` de ces fonctions. Les fonctions utilisateur n'ont donc qu'à s'occuper des signes des quantités.

En ce qui concerne la relation de Bezout, la fonction **bezout** appliquée à deux entiers x et y rend une liste de trois entiers, le pgcd p et deux entiers tels que $ux + vy = p$. On sait que la procédure interne `nbezout` rend des nombres non signés, u et p. En premier lieu on transforme p en nombre positif, u en nombre signé et on calcule v par cette relation. Par ailleurs cet algorithme suppose x et y positifs. Pour ce faire, on l'appelle avec les valeurs absolues de x et y. Le cas où l'un des deux arguments x et y est 0 est trivial. Dans le cas où les deux arguments sont nuls, par convention, on rend 0 pour les trois quantités. Finalement, la fonction interne suppose $x > y$. Si $x < y$ il suffit d'échanger x et y, puis u et v à la fin du calcul. Et si $x = y$, le pgcd est x, et on a le choix entre $u = 0, v = 1$ ou $u = 1, v = 0$. Ceci est le seul cas où les quantités u, v et p ne sont pas uniques sous les hypothèses $p > 0$, $|u| < |y|/p$ et $|v| < |y|/p$.

Algorithme 22. (Bezout) *Les paramètres a et b sont des nombres quelconques. On teste en général qu'ils sont entiers. À partir du point 4 de l'algorithme de Bezout, ce sont des entiers internes, de même que U et p.*

Fonction bezout. *Arguments a et b.*

1. *Appeler* `test_lor_land`(a, b)
2. *Mettre dans s_a et s_b les signes de a et b, qui sont dans* `sign1` *et* `sign2`.
3. *Mettre dans a et b les valeurs absolues de a et b qui sont dans* `arg1` *et* `arg2`.
4. *Comparer a et b via* `ncmp`. *Si $a < b$, échanger a et b, poser $t = 1$, sinon $t = 0$.*
5. *Si $b = 0$ ou $a = b$,*
   - **5.1.** *Si $t = 1$, rendre la liste $(a, 0, c)$, où c est le signe de b, i.e. 1 si s_b est vrai, -1 sinon.*
   - **5.2.** *Si $t = 0$, rendre la liste $(a, c, 0)$ où c est le signe de a.*
6. *Soit $p =$* `nbezout`(a, b), *s la variable globale M, U la variable globale U.*
7. *Si s est pair, mettre faux sinon vrai dans* `sign1`. *Poser $u =$* `zx`(U).
8. *Multiplier U par a. Si s est pair, lui ajouter p, sinon, lui soustraire p. Diviser U par b.*
9. *Si s est pair, mettre vrai sinon faux dans* `sign1`. *Poser $v =$* `zx`(U).
10. *Poser $p =$* `nx`(p).
11. *Si $t = 1$, échanger u et v.*
12. *Si s_a est faux, changer le signe de u, si s_b est faux, changer le signe de v.*
13. *Rendre la liste (p, u, v).*

Fonction modp. *Arguments a et b.*

1. *Erreur si b n'est pas un entier > 1.*
2. *Si a est un entier, rendre le reste de la division de a par b.*
3. *Si a n'est pas une fraction rationnelle, erreur.*
4. *Soient n et d les numérateur et dénominateur de a.*
5. *Appeler* `bezout` *sur d et b.*
6. *Soit (p, u, v) le résultat. Erreur si $p \neq 1$.*
7. *Rendre le reste de la division de nu par b.*

4.4 Fonctions logiques

Nous avons étendu les fonctions logiques à des nombres de taille arbitraire. Pour ce faire, on considère que les nombres négatifs sont représentés en complément à 2.

Un nombre positif doit être considéré comme une suite infinie à gauche de bits, dont seul un nombre fini sont distincts de 0, donc $\dots 0 \dots 0 x_n \dots x_0$ qui correspond au nombre $\sum_{i=0}^{n} 2^i x_i$. Le complément à 1 d'un nombre est obtenu en échangeant les bits 0 et 1. Le complément à 2 est alors obtenu en additionnant 1 au résultat. Par conséquent, si le nombre se termine par k bits nuls, on laisse les $k+1$ derniers bits inchangés, et on échange les 0 et les 1 pour les autres bits. Un nombre négatif est donc une suite infinie de bits dont seul un nombre fini sont différents de 1. La fonction `compl2` calcule le complément à 2 (tronqué sur n mots) d'un nombre en travaillant sur les mots de la façon suivante: en cherche le premier mot non nul en partant de la droite. On prend son complément à 2 en prenant son opposé (la représentation interne de Lisp est une représentation en complément à deux). Pour les autres mots on prend leur complément à 1 en soustrayant 1 du complément à 2.

La fonction **lsh** prend deux arguments x et y. Elle décale x de y positions. Cette fonction est simple: elle rend un objet qui a même signe que x et regarde la valeur absolue. Dans le cas où $y > 0$, le résultat est $x2^y$, sinon c'est la partie entière du quotient de x par 2^{-y}. C'est une erreur si $y > 2^{32}$ sinon on écrit $y = 32a + b$. Il s'agit de décaler x de a mots et de b bits. Le décalage en mots se fait par recopie du vecteur dans un vecteur plus grand ou plus petit, le décalage en bits se fait en utilisant les macros de décalages définies au chapitre 2. Ce décalage peut être vers la gauche ou la droite, indépendamment du signe de y. Par exemple, si $y = 1$, on décale en général de une position vers la gauche. Dans le cas où il y aurait un dépassement de capacité, on copie x dans un vecteur un peu plus grand, avec un zéro à droite, et on décale tout de 31 vers la droite. Dans le cas $y = -1$, on décale en général de une position vers la droite, mais dans le cas où le résultat serait plus petit, on copie x dans un vecteur un peu plus petit, et on décale tout de 31 vers la gauche. Il faut dans ce cas tenir compte des bits de x qui ont disparus dans la copie.

Les opérations **land**, **lor** et **lxor** calculent le *et* logique, *ou* logique ou le *ou exclusif*. Elles utilisent des fonctions internes qui supposent que leur deux arguments sont des vecteurs de même taille, et utilisent les primitives C. Nous ne décrirons pas ici ces trois fonctions, uniquement la fonction **land** (le principe étant le même pour ces trois fonctions).

Soit à calculer le *et* logique entre deux entiers x et y. Dans un premier cas les deux nombres sont positifs. Le résultat est alors un nombre positif, dont la taille est au plus la taille du plus petit des deux nombres. Il suffit donc de copier le plus grand, et d'appeler la fonction interne sur cette copie. Il faudra éventuellement supprimer les zéros en tête. Notons que si l'un des deux arguments est un chiffre, on n'a pas besoin de copier quoi que ce soit. Dans un deuxième cas, l'un des arguments est négatif (par symétrie, on peut supposer que c'est y) et l'autre est positif. Il faut prendre le complément à 2 de y, et le résultat est positif. Finalement si les deux nombres sont négatifs, on prend le complément à deux de chaque nombre, et on réalise l'opération interne, et comme le résultat est négatif, on prend le complément à 2 du résultat. En règle générale, on copie les deux vecteurs dans des vecteurs qui ont comme taille la plus grande des deux tailles, avant de calculer les compléments à 2.

Finalement, la fonction **haulong** donne le nombre de bits de la valeur absolue d'un entier.

Algorithme 23. (Fonctions logiques) *On utilise des entiers x, y, des chiffres n, m, c, c_1, c_2, z, p, d, d_1, d_2, des tableaux de chiffres h, h' des indices i, j, k, s, S, t,*

Les quantités x et y sont aussi des nombres internes.

Procédure test_lor_land. *Arguments x et y.*

1. *Si x n'est pas un entier, erreur.*
2. *Si x est un grand entier, mettre le champ signe de x dans* `sign1` *et sa valeur absolue dans* `arg1`.
3. *Si x est un petit entier, soit $n =$* `Int_val`(x). *Si $n > 0$, mettre vrai dans* `sign1` *et x dans* `arg1`, *sinon mettre faux dans* `sign1` *et* `make_int`$(-n)$ *dans* `arg1`.
4. *Faire de même pour y avec* `sign2` *et* `arg2`.

Fonction lsh. *Arguments x et y.* [Décalage arithmétique x de y. Si $y > 0$, rend $x2^y$, si $x > 0$, rend la partie entière de $x2^y$, sinon l'opposé de la partie entière de $-x2^y$.]

1. *Appeler* `test_lor_land`.
2. *Si $x = 0$ ou $y = 0$, rendre x.*
3. *Si y n'est pas un petit entier, erreur, sinon poser $n =$* `Int_val`(y).
4. *Si $n > 0$*
   - **4.1.** *Écrire $n = 32n' + m$, puis $n = n'$. Mettre* `arg1` *dans x.*
   - **4.2.** *Si x est un chiffre, soit $c =$* `Int_val`(x), *sinon $c = x_0$. Si x est un chiffre, soit $S = 1$, sinon $S = t(x)$.*
   - **4.2.** *Si $m = 0$, poser $c = 0$, sinon $c = c/2^{32-m}$.*
   - **4.3.** *Poser $s = 0$. Si c est non nul, poser $s = 1$, et incrémenter n.*
   - **4.4.** *Si x est un chiffre et $n = 0$, soit $z =$* `Int_val`(x), *multiplier z par 2^m (décalage), rendre* `zx`(`make_int`(z)).
   - **4.5.** *Si x est un chiffre, allouer un vecteur de taille $n + 1$, et y mettre comme premier chiffre* `Int_val`(x) *; appeler x ce vecteur. Sinon, copier x cadré à gauche dans un vecteur de taille $S + n$, via* `acopyvector2` [on a décalé de $32n$ bits.]
   - **4.6.** *Si m est non nul, soit h le pointeur de tas de x.*
     Si $s = 1$, appeler `shift_right`$(h, 0, S, c, 32 - m)$ *sinon* `shift_left`$(h, 0, S - 1, c, m)$.
   - **4.7.** *Rendre* `zx`(x).
5. *Si $n < 0$*
   - **5.1.** *Écrire $-n = 32n + m$ par division.*
   - **5.2.** *Poser $x =$* `arg1`. *Si x est un chiffre, soit $c =$* `Int_val`(x), *remplacer n par $1 - n$, sinon poser $c = x_0$, remplacer n par $t(x) - n$.*
   - **5.3.** *Diviser c par 2^m. Si le résultat est 0, décrémenter n, poser $s = 1$, sinon poser $s = 0$.*
   - **5.4.** *Si $n \leq 0$, rendre 0.*
   - **5.5.** *Si s est vrai, soit $p = x_n/2^m$* [x n'est pas un chiffre.]
   - **5.6.** *Si $n = 1$: si $s = 1$, si $m = 0$ poser $c = p$ sinon $c = x_0 2^{32-m} + p$ et sinon (cas $s \neq 1$) ne rien faire. Rendre* `zx`(`make_int`(c)).
   - **5.7.** *Copier x dans un vecteur de taille n, en utilisant* `acopyvector2`.
   - **5.8.** *Si m est non nul, soit h le pointeur de tas de x.*
     Si $s = 1$, appeler `shift_left`$(h, 0, n - 1, c, 32 - m)$ *sinon* `shift_right`$(h, 0, n - 1, c, m)$. *Si $s = 1$ ajouter p à h_n.*

5.9. *Rendre* `zx`(x).

Procédure compl2. *Argument* x.

1. *Soit* $t = t(x)$, h *le pointeur de tas de* x, $j =$ `afind0_end`$(h, 0, t)$.
2. *Remplacer* h_j *par* $-h_j$.
3. *Pour* $i < j$, *remplacer* h_i *par* $-h_i - 1$.

Procédure log_cp. *Arguments* x, i *et* k.

1. *Si* x *est un chiffre, soit* $c =$ `Int_val`(x), *allouer un vecteur de taille* k, *y mettre* c *en dernière position, rendre le vecteur.*
2. *Dans les autres cas rendre* `acopyvector3`$(x, k, k - i)$.

Macro ilogical. *Argument* x, y, i, j, f [f est l'un de *et*, *ou*, *ou exclusif*].

1. *Soient* h *et* h' *les pointeurs de tas de* x *et* y.
2. *Pour* i *avec* $0 \leq i < j$, *remplacer* h_i *par* $f(h_i, h'_i)$.

Macro last_word. *Argument* x.

Soit $t = t(x)$. *Rendre* x_{t-1}.

Macro log_mem_alloc. *Pas d'arguments.*

1. *Si* `arg1` *est un chiffre, poser* $i = 1$, *sinon mettre dans* i *la taille de* `arg1`.
2. *Mettre dans* j *la taille de* `arg2` *calculée de même.*
3. *Soit* $k = 1 + \max(i, j)$.
4. *Copier les deux vecteurs via* `log_cp`$($`arg1`$, i, k)$ *et* `log_cp`$($`arg2`$, j, k)$.
5. *Appeler* `compl2` *sur* `arg2`.

Fonction land. *Arguments* x *et* y. [Et logique entre x et y.]

1. *Appeler* `test_lor_land`. *Mettre dans* x *et* y *les valeurs absolues qui sont dans* `arg1` *et* `arg2`.
2. *Si* $x = 0$ *ou* $y = 0$, *rendre 0.*
3. *Si* $x > 0$ *et* $y > 0$
   - **3.1.** *Si* x *est un chiffre, soit* $c_1 =$ `Int_val`(x). *Si* y *est un chiffre, soit* $c_2 =$ `Int_val`(y), *sinon* $c_2 =$ `last_word`(y). *Remplacer* c_1 *par le* et *logique entre* c_1 *et* c_2. *Rendre* `nx`$($`make_int`$(c_1))$.
   - **3.2.** *Si* y *est un chiffre, soit* $c_2 =$ `Int_val`(y). *Soit* $c_1 =$ `last_word`(x). *Remplacer* c_1 *par le* et *logique entre* c_1 *et* c_2. *Rendre* `nx`$($`make_int`$(c_1))$.
   - **3.3.** *Si* y *est plus long que* x, *échanger* x *et* y. *Soit* $i = t(x)$, $j = t(y)$.
   - **3.4.** *Copier* x *via* `acopyvector4`$(x, i - j, j)$.
   - **3.5.** *Appeler* `ilogical` *sur* x, y, i, j *et* et.
   - **3.6.** *Rendre* `nx`$($`nunderflow1`$(x))$.
4. *Si* $x < 0$ *et* $y < 0$
   - **4.1.** *Appeler* `log_mem_alloc`.

4.2. *Appeler* `compl2` *sur* `arg1`.

4.3. *Appeler* `ilogical` *sur* `arg1`, `arg2`, i, k *et* et.

4.4. *Appeler* `compl2` *sur* `arg1`.

4.5. *Mettre* `sign1` *à faux, rendre* `zx(nunderflow1(arg1))`.

5. *Si x et y ont des signes différents*

5.1. *Échanger x et y si $x > 0$.*

5.2. *Si y est un chiffre*

Soit $c_2 =$ `Int_val`(y).

Si x est un chiffre, soit $c_1 =$ `Int_val`(x), *sinon $c_1 =$* `last_word`(x).

Remplacer c_1 par le et *logique entre $-c_1$ et c_2.*

Rendre `nx(make_int`(c_1)`)`.

5.3. *Si x est un chiffre*

Soit $c_1 =$ `Int_val`(x), *$i = t(y)$, c_2 le chiffre de y en position $i - 1$.*

Remplacer c_1 par le et *logique entre $-c_1$ et c_2.*

Copier y, mettre c_1 en position $i - 1$ dans y.

Rendre `nx(nunderflow1`(y)`)`.

5.4. *Dans les autres cas*

Soient i et j les tailles de x et y, et $k = \max(i, j)$.

Copier les vecteurs via `acopyvector3`$(x, k, k - i)$, *ou* $(y, k, k - j)$.

Appeler `compl2`(x).

Appeler `ilogical` *sur x, y, i, k et* et.

Rendre `nx(nunderflow1`(x)`)`.

Fonction lor. *Arguments x et y.* [Ou logique entre x et y.]

1. *Appeler* `test_lor_land`. *Mettre dans x et y les valeurs absolues qui sont dans* `arg1` *et* `arg2`.

2. *Si $x = 0$ ou $y = 0$, rendre le nombre non nul.*

3. *Si x et y sont positifs*

3.1. *Si x est un petit entier*

Poser $c_1 =$ `Int_val`(x).

Si y est un petit entier, soit $c_2 =$ `Int_val`(y), *remplacer c_1 par le ou logique entre c_1 et c_2, rendre* `nx(make_int`(c_1)`)`.

Copier y via `acopyvector1`. *Soit $k = t(y)$.*

Remplacer y_k par le ou logique avec c_1.

Rendre `nx`(y).

3.2. *Si y est un petit entier, échanger les arguments, repartir en* **3.1.**

3.3. *Soit $i = t(x)$, $j = t(y)$. Si $j > i$, échanger i et j, x et y.*

3.4. *Copier x via* `acopyvector2`(x, i).

3.5. *Décrémenter i et j. Soient h et h' les pointeurs de tas de x et y.*

3.6. *Tant que $j \geq 0$, remplacer h_i par le ou logique avec h'_j, décrémenter i et j.*

3.7. *Rendre* `nx(nunderflow1`(x)`)`.

4. *Si x et y sont négatifs*

4.1. *Appeler* `log_mem_alloc`.

4.2. *Appeler* `ilogical` *sur* `arg1`, `arg2`, i, k *et* ou.

4.3. *Appeler* `compl2` *sur* `arg1`.

4.4. *Mettre* `sign1` *à faux, rendre* `zx(nunderflow1(arg1))`.

5. *Si x est positif échanger* `arg1` *et* `arg2`.

6. [On a $x < 0$ et $y > 0$].

6.1. *Appeler* `log_mem_alloc`.

6.2. *Appeler* `compl2` *sur* `arg2`.

6.3. *Appeler* `ilogical` *sur* `arg1`, `arg2`, i, k *et* ou.

6.4. *Appeler* `compl2` *sur* `arg1`.

6.5. *Mettre* `sign1` *à faux, rendre* `zx(nunderflow1(arg1))`.

Fonction lxor. *Arguments x et y.* [Ou exclusif logique entre x et y.]

1. *Appeler* `test_lor_land`. *Mettre dans x et y les valeurs absolues qui sont dans* `arg1` *et* `arg2`.

2. *Si $x = 0$ ou $y = 0$, rendre le nombre non nul.*

3. *Si x et y sont positifs*

3.1. *Si x est un petit entier*

Poser $c_1 =$ `Int_val`(x).

Si y est un petit entier, soit $c_2 =$ `Int_val`(y), *remplacer c_1 par le ou exclusif logique entre c_1 et c_2, rendre* `nx(make_int`(c_1)`)`.

Copier y via `acopyvector1`. *Soit $k = t(y)$.*

Remplacer y_k par le ou exclusif logique avec c_1.

Rendre `nx`(y).

3.2. *Si y est un petit entier, échanger les arguments, repartir en* **3.1.**

3.3. *Soit $i = t(x)$, $j = t(y)$. Si $j > i$, échanger i et j, x et y.*

3.4. *Copier x via* `acopyvector2`(x, i).

3.5. *Décrémenter i et j. Soient h et h' les pointeurs de tas de x et y.*

3.6. *Tant que $j \geq 0$, remplacer h_i par le ou exclusif logique avec h'_j, décrémenter i et j.*

3.7. *Rendre* `nx(nunderflow1`(x)`)`.

4. *Si x et y sont négatifs*

4.1. *Appeler* `log_mem_alloc`.

4.2. *Appeler* `ilogical` *sur* `arg1`, `arg2`, i, k *et* ou exclusif.

4.3. *Appeler* `compl2` *sur* `arg1`.

4.4. *Mettre* `sign1` *à faux, rendre* `zx(nunderflow1(arg1))`.

5. *Si x est positif échanger* `arg1` *et* `arg2`.

6. [On a $x < 0$ et $y > 0$].

6.1. *Appeler* `log_mem_alloc`.

6.2. *Appeler* `compl2` *sur* `arg2`.

6.3. *Appeler* `ilogical` *sur* `arg1`, `arg2`, i, k *et* ou exclusif.

6.4. *Appeler* `compl2` *sur* `arg1`.

6.5. *Rendre* `nx`(`nunderflow1`(`arg1`)).

Fonction lnot. *Argument* x. [Négation logique de x.]

1. *Rendre* `lxor`$(x, -1)$.

Fonction power-of-two. *Argument* x. [Calcule 2^x.]

1. *Si* x *n'est pas un petit entier, erreur.*

2. *Soit* $d =$ `Int_val`(x). *Si* $d > 0$ *rendre* `lsh`$(1, x)$.

3. *Sinon, remplacer* x *par* $-x$, *rendre* $1/$`lsh`$(1, x)$.

Procédure haulongfix. *Arguments* i *et* n.

Si $n = 0$ *rendre* i.

Sinon rendre `haulongfix`$(i + 1, n/2)$.

Procédure ihaulong. *Argument* x.

1. *Si* x *est un vecteur, poser* $d_1 = 32(t(x) - 1)$ *sinon* $d_1 = 0$.

2. *Si* x *est un chiffre, soit* $c =$ `Int_val`(x) *sinon* $c = x_0$.

3. *Si* $c < 0$ *poser* $d_2 = 32$ *sinon* $d_2 =$ `haulongfix`$(c, 0)$.

4. *Rendre* `make_int`$(d_1 + d_2)$.

Fonction haulong. *Argument* x. [Rend z avec $2^{z-1} < |x| \leq 2^z$]

1. *Si* x *est un petit entier*

Soit $c =$ `Int_val`(x).

Si $c < 0$, *poser* $c = -c$.

Rendre `make_int`(`haulongfix`$(0, c)$).

2. *Si* x *est un grand entier, rendre* `ihaulong`(z) *où* z *est le champ* `rz_num` *de* x.

3. *Si* x *n'est pas une fraction, erreur.*

4. *Remplacer* x *par sa valeur absolue.*

5. *Soit* c *la différence de* `ihaulong` *appliqué au numérateur et au dénominateur de* x. *Soit* $y =$ `power-of-two`(`make_int`(c)).

6. *Si* $x < y$, *incrémenter* c.

7. *Rendre* `make_int`(c).

4.5 Puissance modulaire

L'algorithme qui suit calcule $a^b \bmod c$. Le résultat est un nombre positif, et plus petit que c. En particulier b peut être très grand. L'algorithme utilisé consiste à prendre une variable intermédiaire p, et à remplacer p par ap si b est impair, à remplacer a par a^2, puis à diviser b par 2, tant que $b \neq 1$. Le résultat est alors ap. On suppose $c > 0$, et $b \geq 0$. Dans le cas $b < 0$, on commence par

inverser a modulo c, grâce à la fonction **modp**. On utilise cette fonction de manière systématique, de telle sorte que a peut être un rationnel quelconque.

Comme b peut être très grand, on évite d'allouer de la mémoire chaque fois qu'on divise par 2. L'algorithme est le suivant: Dans le cas où b est un vecteur de chiffre, on met dans s l'indice du dernier chiffre, et on positionne p à 0. On teste si le bit en position p du chiffre s est nul. On incrémente p; si p dépasse 32, on le remet à 0, et on décrémente s. On teste un bit en faisant le et logique entre le chiffre et 2^p. La quantité 2^{p+1} est déduite de 2^p par décalage. Dans le cas où b a un seul chiffre, ou si on regarde le premier chiffre, on met ce chiffre dans la variable y. L'algorithme devient alors: tant que $y \neq 1$, tester la parité de y, diviser y par 2. Notons que si b a plus d'un chiffre, la quantité y est initialisée à 2. Le test d'arrêt est donc indépendant de s.

La boucle principale consiste à calculer xp modulo c. On suppose $0 \leq x < c$ et $0 \leq p < c$. Dans le cas où c tient sur 16bits, on multiplie et on divise en utilisant l'arithmétique de C. Dans le cas où c est un chiffre, la multiplication se fait par `ex*` et la division par `ex/`.

Dans le cas général, on alloue deux vecteurs x et p, et un vecteur tampon Z. Les deux premiers vecteurs ont une taille s, le dernier une taille $2s$. On calcule le produit dans Z, et on divise. Notons que pour diviser, il faut multiplier Z et c par un facteur de normalisation z. On précalcule zc dans c'. Le reste de la division, qui est la quantité qui nous intéresse doit être divisé par z; mais ceci est fait par la procédure `iquomod8t`. Ce reste est alors copié dans x ou p.

L'algorithme de division suppose $X \geq c$. Il faut le tester. Comme X est cadré à droite, on ne peut pas utiliser `ncmp`. Pour simplifier un peu la gestion des indices, les vecteurs x et p ont dans leur premier mot l'indice du premier chiffre utile. La procédure est donc un peu compliquée, mais la taille mémoire allouée est 4 fois celle de c, plus une copie éventuelle de a si $a \geq c$, plus une copie du résultat (il faut au moins supprimer le premier mot de x). Notons que si x devient nul, on arrête l'algorithme de suite.

Remarque: dans le cas où on calcule $a^b \bmod c$, et que b est très grand devant c, on peut optimiser comme suit. On suppose c petit, disons $c < 2^{16}$, de sorte qu'on peut le factoriser sans faire appel à `powermod`. On peut toujours écrire $a = a_1 a_2$ où a_1 est premier à c, et tous les facteurs premiers de a_2 divisent c, en ne calculant que des pgcds. Il suffit de calculer $a_1^b \bmod c$ et $a_2^b \bmod c$, et de rendre le produit des deux quantités modulo c. Dans le cas où a est premier à c, si $b = q\phi(c) + r$, alors $a^b = a^r \pmod c$. La factorisation de c est utilisée pour calculer $\phi(c)$. Dans le cas où tous les facteurs de a divisent c, écrivons $c = c_1 c_2$, où c_2 est premier à a, et a et c_1 ont même support premier. En particulier c_1 et c_2 sont premiers entre eux, il existe u et v avec $uc_1 + vc_2 = 1$. Alors $(X \bmod c_2)uc_1 + (X \bmod c_2)vc_1$ est égal à X modulo c. Dans le cas $X = a^b$, le terme $X \bmod c_2$ se calcule comme précédemment, et $X \bmod c_1$ est nul si b est assez grand (il suffit que b soit plus grand que le plus grand exposant dans c_1). Cet algorithme n'est pas implémenté, mais pourrait l'être.

Algorithme 24. (Puissance modulaire) *On utilise ici des nombres a, b, c, des chiffres m, r, I, w, m, x, y, n, des tableaux de chiffres X, Y, N, C, A, B, C', Z des indices j, z. On utilise également une variable globale P, le tampon de Karatsuba.*

Fonction powermod. *Arguments a, b et c.*

1. *Erreur si b et c ne sont pas des entiers, ou si $c \leq 1$, ou si a n'est pas un rationnel.*
2. *Si $b < 0$, remplacer b par $-b$, a par $1/a$.*
3. *Si $a = 0$ ou $a = 1$ rendre a.*

4. *Si* $b = 0$ *rendre* 1.
5. *Réduire* a *modulo* c, *via* `modp`.
6. *Si* $a = 0$ *ou* $a = 1$ *ou* $b = 1$, *rendre* a.
7. *Si* $b = 2$, *rendre* a^2 *modulo* c.
8. *Si* b *est un grand entier, le remplacer par son champ* `rz_num`.
9. *Si* c *est un petit entier, rendre* `power_mod_fix`(a, b, c).
10. *Remplacer* c *par son champ* `rz_num`.
11. *Si* c *est un petit entier : si* a *n'est pas un petit entier, remplacer* a *par son champ* `rz_num`. *Rendre* `power_mod_fix`(a, b, c).
12. *Rendre* `power_zn3`(a, b, c).

Macro init_divide. *Pas d'arguments.*

1. *Poser* $m = 1$, $y = 2$, $w = 0$, $r = 0$, $I = 0$, $s = 0$.
2. *Si* Y *est un chiffre, mettre* `Int_val`(Y) *dans* y.
3. *Sinon, poser* $s = t(Y) - 1$, $w = Y_s$.

Macro divide2. *Pas d'arguments.*

1. *Si* $s = 0$, *mettre 1 dans* r *si* y *est impair, 0 sinon. Remplacer* y *par* $y/2$. *Fin de la macro.*
2. *Mettre 0 dans* r *si le 'et' logique entre* w *et* m *est 0,* y *mettre 1 sinon.*
3. *Incrémenter* I, *multiplier* m *par 2 (décalage).*
4. *Si* $I \neq 32$, *fin de la macro.*
5. *Décrémenter* s, *mettre* Y_s *dans* w, *poser* $m = 1$, $I = 0$.
6. *Si* $s = 0$, *mettre* w *dans* y.

Procédure power_mod_fix1. *Arguments* x, Y, n.

1. *Poser* $p = 1$. *Appeler* `init_divide`.
2. *Tant que* $y \neq 1$,

   Appeler `divide2`.
   Si $r = 1$, *remplacer* p *par* $px \bmod n$.
   Remplacer x *par* $x^2 \bmod n$.
   Si $x = 0$, *rendre 0.*
3. *Remplacer* x *par* $px \bmod n$.
4. *Rendre* `nx`(`make_int`(x)).

Procédure power_mod_fix. *Arguments* X, Y, N.

1. *Poser* $x =$ `Int_val`(X), $n =$ `Int_val`(N).
2. *Si* $n < 2^{16}$, *rendre* `power_mod_fix1`(x, Y, n).
3. *Sinon, même code que pour* `power_mod_fix1`, *sauf que* $ab \bmod c$ *est calculé en mettant* `C` *à 0, en calculant* `ex/`(`ex*`$(a, b, 0), c)$, *et le résultat est* `C`.

Procédure mul_less. *Arguments* X, C, i_x, t_x *et* t_c. [rend vrai si $x < c$, le vecteur c est pris entre 0 et t_c, et x entre i_x et t_x.]

1. *Si* $t_x - i_x < t_c$, *rendre vrai.*
2. *Si* $t_x - i_x > t_c$ *rendre faux.*
3. *Pour* $i = 0$, $j = i_x$, *tant que* $i \leq t_c$, *si* $X_i = C_i$, *incrémenter* i *et* j.
4. *Si* $i > t_c$ *ou* $X_j > C_i$ *rendre faux, sinon vrai.*

Procédure mul_zn3. *Arguments* A, B, C, z, C', Z, t_a *et* t_z. [Calcule $ab \bmod c'$. Le facteur de normalisation est z, et $c = c'2^z$; t_a est la taille de a et b, $t_a - 2$ est la taille de c et c'. On utilise un vecteur auxiliaire Z. Les arguments A, B, C et C' sont les pointeurs de tas de a, b, c et c'. L'indice du premier chiffre utile de a ou b est dans la première case du vecteur.]

1. *Remplir le vecteur* Z *avec des zéros.*
2. *Mettre dans* i_A *et* i_B *les premiers chiffres de* B *et* A, *dans* t_A *et* t_B *la quantité* $t_a - 1$, *mettre* $t_z - 1$ *dans* t_C.
3. *Appeler* `itimes0` *sur* A, B, Z *et* P.
4. *Appeler* `mul_less`$(Z, C', i_C, t_C, t_a - 3)$. *Si le résultat est vrai, poser* $i_B = i_C$, $t_B = t_C$.
5. *Sinon*
   - **5.1.** *Poser* $i_A = i_C$, $t_A = t_C$, $i_B = 0$, $t_B = t_a - 3$.
   - **5.2.** *Si* $z \neq 0$, *appeler* `shift_left` *sur* Z, i_A, t_A, w *et* z. *Si* w *est non nul, décrémenter* i_A, *mettre* w *dans* Z *en position* i_A.
   - **5.3.** *Appeler* `iquomod8t` *sur* Z, C *et sur* z.
6. *Poser* $w = t_B - i_B + 1$, $z = t_a - w$.
7. *Copier* Z *dans* A, *en partant de* z *pour* A *et* i_B *pour* Z.
8. *Mettre* z *en position 0 dans* a.

Procédure power_zn3. *Arguments* a, Y *et* n. [On notera H_x le pointeur de tas de x.]

1. *Poser* $s = t(n) + 1$, $N = 2(s + 1)$.
2. *Mettre dans* Z *un vecteur de taille* N, *dans* P *un vecteur de taille* $2N$.
3. *Poser* $n' = n$, *copier* n *via* `acopyvector1`.
4. *Soit* $z =$ `anormalise`(n_0).
5. *Appeler* `shift_left`$(H_n, 0, s - 1, \alpha, z)$.
6. *Soit* $w = s$; *incrémenter* s.
7. *Allouer un vecteur* p *de taille* s. Y *mettre* w *en position 0, et 1 en position* w.
8. *Allouer un vecteur* x *de taille* s. *Si* a *est un chiffre, mettre* `Int_val`(a) *en position* w, *sinon poser* $w = s - t(a)$, *copier* a *à partir de 0 dans* x *à partir de* w. *Mettre* w *en position 0 dans* x. *Poser* $a = x$.
9. *Appeler* `init_divide`.
10. *Tant que* $y \neq 1$ [`mul_zn3` est appelé avec 8 paramètres, les derniers sont H_n, z, $H_{n'}$, H_Z, s et N.]

    Appeler `divide2`.

Si $r = 1$, *appeler* `mul_zn3` *sur* H_p *et* H_a.
Appeler `mul_zn3` *sur* H_a *et* H_a.
Si $a = 0$, *rendre 0.*

11. *Appeler* `mul_zn3` *sur* H_a *et* H_p.

12. *Soit* s *le premier chiffre de* a.

13. *Rendre* `nx(nunderflow4`$(a, s, t(a) - 1)$`)`.

4.6 Racines

Pour trouver la racine p^{e} d'un nombre x, on applique la méthode de Newton pour trouver la racine de $g(b) = b^p - x$. Ceci donne la relation de récurrence:

$$b_{n+1} = b_n - \frac{b_n^p - x}{p b_n^{p-1}}. \tag{0}$$

Soit

$$F(b) = b - \frac{1}{p}(b - \frac{x}{b^{p-1}}) = \frac{(p-1)b^p + x}{pb^{p-1}} \tag{1}$$

$$G(b) = b + \left\lfloor \frac{1}{p} \left(\lfloor \frac{x}{b^{p-1}} \rfloor - b \right) \right\rfloor. \tag{2}$$

On notera $x = c^p$, et on cherche d, la partie entière de c. La méthode de Newton consiste à considérer la suite $b_{i+1} = F(b_i)$. Si la suite a une limite, c'est c et la convergence est quadratique. Comme on cherche un résultat entier, on utilise la récurrence $b_{n+1} = G(b_n)$. Notons $y = \lfloor x/b^{p-1} \rfloor$. Dans le cas $p = 2$, c'est juste $\lfloor x/2 \rfloor$ et $G(b) = \lfloor (y + b)/2 \rfloor$.

Écrivons $x/b^{p-1} = y + \alpha$, avec $0 \leq \alpha < 1$, et $(y - b)/p = v + \beta$, avec $0 \leq \beta < 1$. Comme y, b et p sont entiers, on a en fait $0 \leq \beta \leq 1 - 1/p$. On a $F(b) = G(b) + \beta + \alpha/p$ d'où $G(b)$ est la partie entière de $F(b)$. On a donc

$$F(b) - 1 < G(b) \leq F(b). \tag{3}$$

On a $F(b) - c = (b/p)[p - 1 + (c/b)^p - pc/b]$. Soit $f(x) = p - 1 + x^p - px$. On a $f'(x) = p(x^{p-1} - 1)$ et $f''(x) = p(p-1)x^{p-2}$. Pour $x \geq 0$ on a $f''(x) \geq 0$, et $f'(1) = 0$, et la fonction f a un minimum en $x = 1$. On en déduit $f(x) \geq 0$ pour $x \geq 0$, donc $F(b) \geq c$. On en déduit $d - 1 \leq c - 1 \leq F(b) - 1 < G(b)$. Comme d et $G(b)$ sont entiers, on en déduit

$$d \leq G(b). \tag{4}$$

Cette relation nous dit d'abord que $b_n > 0$ pour tout n, et y est le quotient entier de x par b^{p-1}. Supposons maintenant $G(b) \geq b$. Cette condition est équivalente à $y \geq b$, d'où $x/b^{p-1} \geq b$, donc $c \geq b$ et donc $b \leq d$. Ceci nous dit alors: dans la suite $b_{n+1} = G(b_n)$, on a $b_n \geq d$ (sauf éventuellement $n = 0$). Calculons b_{n+1} tant que $b_{n+1} < b_n$. Supposons $b_{n+1} \geq b_n$. Alors $b_n \leq d$, mais $b_n \geq d$. On en déduit que b_n est le résultat cherché.

Soit $L = L(x)$ tel que $2^L \leq x < 2^{L+1}$. Notons que `haulong`(x) est $L + 1$. Écrivons $L = pq + r$ par division. Soit $z = (r + 1)/p$. On a $0 \leq z \leq 1$ et $2^z \leq z + 1$; ce dernier résultat se montre comme suit: si $g(z) = 2^z - z - 1$, $g''(z) > 0$, $g(0) = g(1) = 0$, donc g est négative entre 0 et 1.

Soit $b_0 = \lfloor 2^q(p+1+r)/p \rfloor$. On a $b_0 \leq 2^{q+1}$, donc $b_0 \leq 2c$. De plus, $c < 2^{(L+1)p} = 2^{q+z} \leq 2^q(1+z)$, donc l'entier d qui satisfait $d \leq c$ satisfait $d \leq \lfloor 2^q(1+z) \rfloor = b_0$. On en déduit $d \leq b_0 \leq 2d$. On ne peut pas trouver de meilleure condition initiale si on ne connait que la taille L de x.

Soit maintenant $W(x)$ la fonction définie par

- On définit L, q, r comme précédemment.
- Soit X tel que $2^{2X} \leq p < 2^{2X+2}$, $\alpha = \lfloor q/2 - X \rfloor$.
- Si $q \leq 16$, ou $\alpha \leq 4$, soit $b_0 = \lfloor 2^q(p+1+r)/p \rfloor$, $b_{n+1} = G(b_n)$. Dans ce cas, $W(x)$ est le premier b_n tel que $b_{n+1} \geq b_n$.
- Sinon, soit $\beta = \lfloor x/2^{\alpha p} \rfloor$, $U = 2^\alpha W(\beta)$. Alors $W(x) = G(U)$.

Dans l'algorithme qui suit, X est noté x_s, c'est une quantité qui ne dépend que de p. On a $L(\beta) = L(x) - \alpha p$. Finalement, $G(U)$ est calculé comme étant la partie entière de $F(U)$.

On prétend que $W(x) = d$ ou $W(x) = d+1$. Ceci est clair si $\alpha \leq 4$, par ce qu'on vient de montrer plus haut, car $W(x) = d$. Si $v = G(U)$, il suffit de montrer que $v \leq d+1$, car on sait $v \geq d$. Il suffit de montrer que $F(U) \leq c+1$ car cela implique $v = G(U) \leq F(U) \leq c+1 < d+2$, donc $v \leq d+1$. Par récurrence on a

$$(\frac{U}{2^\alpha} - 1)^p \leq \beta \leq (\frac{U}{2^\alpha} + 1)^p$$

donc

$$(U - 2^\alpha)^p \leq c^p \leq (U + 2^\alpha)^p + 2^{\alpha p}. \tag{5}$$

Montrons d'abord $U \geq p2^\alpha$. Comme $\alpha \geq 4$, on a $q/2 - X \geq 4$ donc $q \geq 2X + q/2 - X + 4 \geq 2X + \alpha + 3$ et donc $q - \alpha \geq 2X + 3$. On a donc

$$\beta \geq \lfloor 2^{L-\alpha p} \rfloor = \lfloor 2^{p(q-\alpha)+r} \rfloor \geq 2^{p(q-\alpha)} \geq 2.2^{p(2X+2)} \geq 2.2^p.$$

On en déduit $W(\beta) \geq p$, donc $U \geq p2^\alpha$.

Montrons maintenant $c/2 \leq U \leq cp/(p-1)$. Par (5), on a $U - 2^\alpha \leq c$, donc $U - c \leq 2^\alpha \leq U/p$. d'où la seconde inégalité. La deuxième inégalité de (5) donne $c^p \leq (U + 2^\alpha)^p + 2^{\alpha p} \leq 2^p U^p$, donc $c \leq 2U$. En fait, si $w = 2^\alpha/U$, il faut montrer $(1+w)^p + w^p \leq 2^p$. Une telle relation est vraie si $w \leq 1/2$. Comme on sait $w \leq 1/p$, elle est vraie pour $p \geq 2$.

Montrons finalement $F(U) \leq c+1$. Posons $z = c/U$. Il s'agit de montrer

$$p - 1 + z^p \leq pz + pz^{p-1}.$$

sachant $(p-1)/p \leq z \leq 2$. Soit $g(z) = z^p + p - 1 - pz - pz^{p-1}$.

On a $g''(z) = p(p-1)[z - (p-2)]z^{p-3}$. Supposons $p \geq 4$ pour commencer. Dans l'intervalle considéré, on a $g''(z) \leq 0$. Comme $g'(0) < 0$, la fonction g est décroissante sur $[0,2]$. Il suffit donc de vérifier que pour $z = 1 - 1/p$ on a $g(z) < 0$. Or $g(z) = (z-p)z^{p-1}$. Dans le cas $p = 3$, on a $g'(z) = 3(z^2 - 2z - 1)$, donc g est décroissante sur $[0,2]$, la conclusion est la même. Dans le cas $p = 2$, les racines de g sont $2 \pm \sqrt{3}$, et la conclusion est vraie aussi.

Remarque: supposons $b = c(1+\epsilon)$ où ϵ est petit. Alors $(F(b) - c)/b$ est équivalent à $\epsilon^2(p-1)/2$. Sans le facteur $(p-1)/2$, l'algorithme double le nombre de chiffres exacts. L'algorithme consiste à

calculer un peu plus que la moitié du nombre de chiffres ($L(p)/2$ chiffres en plus) et une itération. Pour avoir k chiffres sur le résultat, il suffit de faire les calculs sur les pk premiers chiffres de x. La contrainte $\alpha \geq 4$ nous dit que au moins 4 bits sont calculés, dans le cas contraire, on utilise la méthode itérative. La contrainte $q \leq 16$ dit que le résultat tient sur 16 bits. Dans ce cas on utilise aussi la méthode itérative. Note: une erreur s'est glissée dans les formules précédentes.

On propose un autre algorithme, plus simple. Il consiste à remplacer G par H avec

$$H(b) = b - \left\lfloor \frac{1}{p}(b - \lfloor \frac{x}{b^{p-1}} \rfloor) \right\rfloor . \tag{6}$$

Il est facile de voir que

$$F(b) - \frac{p-1}{p} \leq H(b) \leq F(b) + \frac{1}{p}$$

Si on compare avec la relation (3), on voit que $H(b) = G(b) + 1$ ou $H(b) = G(b)$, d'où $H(b) \geq d$. Dans ce cas, si $b = b_n = b_{n+1}$ on a

$$b^p - (p-1)b^{p-1} \leq c^p \leq b^p + b^{p-1}$$

donc $c_0 \leq b \leq c_1$, pour certaines quantités c_0 et c_1. On a $c - 1 \leq c_0 \leq c$, et $c \leq c_1 \leq c + p$. Cette dernière approximation peut être rendue meilleure: si c est très grand, $c_1 \leq c + 1$. L'algorithme est alors: calculer $b_{n+1} = H(b_n)$ tant que $b_{n+1} < b_n$. On a alors une bonne approximation de d, qui est calculé par recherche linéaire.

Algorithme 25. (Racines) *Tous les paramètres et variables locales sont des entiers.*

Fonction isqrt. *Argument x.*

1. *Erreur si x n'est pas un entier ≥ 0.*
2. *Rendre x si $x = 0$ ou $x = 1$.*
3. *Soit n le nombre de bits de x. Si n est pair et x est une puissance de 2, rendre $2^{n/2}$* [il faut vérifier que un seul bit est non nul dans x].
4. *Si $x < 100$: comparer x à 4, 9, 16, 25, 36, 49, 64 et 81. Si par exemple $25 \leq x < 36$ rendre 5.*
5. *Soit $q =$* `haulong`(x), *$a = 0$, $b = 2^{q/2}$, $c = b$.*
6. *Faire: $a = (b + x/b)/2$ (prendre la partie entière des quotients via* `quomod`*); si $a = b$, rendre a, si $a = c$, rendre le minimum de a et b, sinon poser $c = b$, $b = a$, continuer la boucle.*

Fonction iroot. *Argument x et n.*

1. *Erreur si n n'est pas un entier ≥ 2, si x n'est pas un entier ≥ 0.*
2. *Si $x = 0$ ou $x = 1$, rendre x.*
3. *Si $n = 2$, rendre* `isqrt`(x).
4. *Poser $a_0 = 0$, $N = n - 1$, $d =$* `haulong`(x), *$d = qn + r$ par division, $a = \lfloor 2^q(n + 1 + r)/n \rfloor$.*
5. *Poser $a = a - (a - x/a^{n-1})/n$, $a_0 = x$.*
5. *Tant que $a < a_0$, poser $a_0 = a$, $a = a - (a - x/a^{n-1})/n$,* [parties entières des quotients, calculées via `quomod`].

6. *Si $a^n \leq x < (a+1)^n$ rendre a. Si la première inégalité est fausse, décrémenter a jusqu'à ce qu'elle devienne vraie. Si la seconde est fausse, incrémenter a jusqu'à ce qu'elle devienne vraie. Rendre alors a.*

Fonction jroot. *Argument x.*

1. *Erreur si n n'est pas un entier ≥ 2, si x n'est pas un entier ≥ 0.*

2. *Si $x = 0$ ou $x = 1$, rendre x.*

3. *Si $n = 2$, rendre* `isqrt`(x).

4. *Soit $a =$* `haulong`$(x) - 1$, $b =$ `haulong`$(n) - 1$. *Poser $a =$* `jroot1`$(x, n, a, b/2)$.

5. *Tant que $a^n > x$ décrémenter a. Rendre a.*

Procédure jroot1. *Arguments n, r, l, x_s.*

1. *Poser $s = \lfloor l/r \rfloor$.*

2. *Poser $x = \lfloor s/2 \rfloor - x_s$.*

3. *Si $s \geq 17$ et $x > 5$*

3.1. *Poser $u =$* `jroot1`$(n/2^{xr}, r, l - rx, x_s)$.

3.2. *Poser $u = u2^x$, $T = u^{r-1}$.*

3.3. *Rendre le quotient de $n + (r-1)uT$ par rT.*

4. *Sinon*

4.1. *Soit f la somme de $r+1$ et du reste de la division de l par r.*

4.2. *Soit u le quotient de $f2^s$ par r.*

4.3. *Poser $x = u$, $u = x + (n/x^{r-1} - x)/r$. Les quotients sont calculés par* `quomod`.

4.4. *Si $u \geq x$, rendre x, dans le cas contraire, continuer en* **4.3.**

4.7 Entrées-sorties

La représentation externe d'un entier est la suite de ses caractères. Le programme d'impression utilise la variable-fonction **obase** pour déterminer la base dans laquelle imprimer les chiffres. Si la base est plus grande que 10, on utilise des lettres majuscules à la place de chiffres, en commençant par la lettre A. Le programme interne d'impression des entiers a été décrit au premier chapitre. La fonction de lecture utilise la variable-fonction **ibase** pour trouver la base. Pour entrer un entier, il suffit de donner la suite de ses chiffres. Notons qu'il ne faut pas donner plus de chiffres que ne peut contenir le tampon interne de lecture de Lisp. Dans notre version, il n'y a plus de limitation sur cette taille, le tampon étant alloué dynamiquement.

Il est possible de préciser la base de lecture au moyen d'une construction de la forme `#12R25` (25 en base 12). Dans ce cas le 12 est lu en base 10. Notons que la documentation LeLisp précise que `#12r25` rend la même résultat. La différence essentielle est que, dans le cas de `#12r25`, si la base n'est pas 10, les calculs sont faits modulo E (donc 2^{16}). En conséquence `#$8000` est égal à $-E/2$. Ce mécanisme ne marche plus dans notre version : il n'y a pas de moyen de créer $-E/2 = -2^{31}$ en tant que petit entier. Une autre différence est que le macro caractère R ignore les espaces et les changements de lignes après un backslash. En d'autres termes, le nombre peut être fourni sur plusieurs lignes (mécanisme analogue à celui de Maple).

Les nombres rationnels sont lus et imprimés sous la forme `1/2`. Il peuvent bien entendu être donnés sous la forme `#12R25/3`.

Il est possible d'imprimer des nombres rationnels de façon différente, en utilisant la fonction **prin-rational**. Elle prend trois arguments, un rationnel x, un type T, et une précision p. Si le type est *float0*, un développement décimal, avec au plus p chiffres sera imprimé, et la période est imprimée avec des accolades. Si le type est *float1*, un développement décimal, avec au plus p chiffres sera imprimé. La période ne sera pas imprimée dans ce cas. Si le type est *cf0*, *cf1* ou *cf-1*, le développement en fraction continue avec au plus p termes sera imprimé. Dans ce dernier cas, si p est négatif, le développement en fraction continue est retourné au lieu d'être imprimé. La fonction **cf-to-rational** au contraire transforme une fraction continue en nombre rationnel.

Les nombres complexes peuvent être lus et imprimés sous la forme `#C(1 2)`. Pour rendre cette notation plus lisible, on peut changer une variable globale de sorte que le résultat imprimé soit `[2i+1]`.

Finalement, pour pouvoir relire rapidement des nombres, si la variable **#:system:print-for-fast-read** et vraie, et si **#:system:print-for-read** est également vraie, les entiers et les fractions sont imprimés dans leur représentation interne, précédée de `#10R/2`. Ici le 2 signifie que c'est deuxième version du logiciel. Dans la première version la représentation interne des vecteurs de chiffres est de la forme `#[a b c ...]`, où a, b, c, etc sont des entiers 16bits signés. Dans la seconde version, c'est `#<abc...>`, où a, b, c sont des chiffres en base 16, il y a assez de zéros en tête pour que le nombre de chiffres soit multiple de 8.

Algorithme 26. (Lecture des nombres) *Certaines fonctions décrites ici sont des fonctions de base. Elles utilisent le tampon interne de Lisp. On notera c_x le code ASCII du caractère x, et b la base de lecture courante.*

Procédure is_a_digit. *Paramètre c, un code ASCII* [Rend vrai si c est le code d'un chiffre dans la base courante].

1. *Si la base est* 10, *rendre vrai si* $c_0 \leq c \leq c_9$.
2. *Si la base est plus petite que* 10, *rendre vrai si* $c_0 \leq c < c_0 + b$.
3. *Sinon, rendre vrai si* $c_0 \leq c \leq c_9$.
4. *Sinon. Si* $c_A \leq c \leq c_Z$, *remplacer* c *par* $c + c_a - c_A$. *Si* $c_a \leq c < c_a + b - 10$, *rendre vrai.*

Procédure tryflo. *Paramètre s, une chaîne de caractères* [Convertit la chaîne en nombre flottant ou en symbole].

1. *Poser* $i = 0$, $n = 0$.
2. *Si* $s_i = c_+$ *ou* $s_i = c_-$, *incrémenter* i.
3. *Tant que* $c_0 \leq s_i \leq c_9$, *incrémenter* i *et* n.
4. *Si* $s_i = c_.$ [un point], *incrémenter* i, *tant que* $c_0 \leq s_i \leq c_9$, *incrémenter* i *et* n.
5. *Si* $n = 0$, *la chaîne n'est pas un nombre, fabriquer un symbole.*
6. *Si* s_i *est la marque de fin de chaîne, utiliser* `atof` *pour convertir la chaîne en flottant. Rendre ce flottant.*
7. *Si* $s_i \neq c_e$ *et* $s_i \neq c_E$, *la chaîne n'est pas un nombre, fabriquer un symbole.*
8. *Incrémenter* i, *poser* $n = 0$.

9. *Répéter* **2** *et* **3.**

10. *Si* $n = 0$, *ou* c_i *n'est pas la marque de fin de chaîne, la chaîne n'est pas un nombre, fabriquer un symbole.*

11. *Utiliser* `atof` *pour convertir la chaîne en flottant. Rendre ce flottant.*

Procédure tryatom. *Paramètre* s, *une chaîne de caractères* [convertit la chaîne en entier, rationnel, flottant ou symbole].

1. *Poser* $i = 0$, $n = 0$.

2. *Si* $s_i = c_+$ *ou* $s_i = c_-$, *incrémenter* i.

3. *Tant que* `is_a_digit`(s_i) *est vrai, incrémenter* i *et* n.

4. *Si* $n = 0$, *rendre* `tryflo`(s).

5. *Si le caractère courant est un slash :*

5.1. *Incrémenter* i, *poser* $n = 0$.

5.2. *Tant que* `is_a_digit`(s_i) *est vrai, incrémenter* i *et* n.

6. *Si* $n > 0$ *et le caractère courant est la marque de fin de chaîne : si la base est* 10, $i < 10$, *et pas de slash : utiliser* `atoi` *pour convertir la chaîne en petit entier. Dans les autres cas, utiliser* `string_to_number`.

7. *Sinon : rendre* `tryflo`(s).

Procédure make_unsigned. *Argument* i [Convertit les chiffres à partir de i du tampon interne en entiers 32bits en base 16].

1. *Poser* $n = 0$.

2. *Pour* j *de* 0 *à* 7, *soit* c *le caractère dans le tampon interne à la position* $i + j$. *Remplacer* n *par* $16n + c - c_0$.

3. *Rendre* n.

Procédure make_address. *Pas d'argument* [Si par exemple le tampon contient `0x1234`, rend l'objet à l'adresse hexadécimale 1234 ; ne vérifie pas que l'adresse est valide].

1. *Poser* $n = 0$.

2. *Pour* $i \geq 2$, *tant que* c *le caractère dans le tampon interne à la position* i *n'est pas la marque de fin de chaîne, remplacer* n *par* $16n + c - c_0$.

3. *Rendre* n, *en tant qu'objet Lisp.*

Procédure make_internal. *Pas d'argument* [Lecture rapide d'un vecteur de chiffres en base 16].

1. *Poser* $i = 0$, $j = 0$, $n = t/8$ [t est la taille du tampon].

2. *Allouer un vecteur de chiffres* V *de taille* n.

3. *Tant que* $n > 0$ *: appeler* `make_unsigned`(i), *mettre le résultat dans le vecteur en position* j, *décrémenter* n, *incrémenter* j, *incrémenter* i *de* 8.

4. *Si* $t = 8$, *rendre* `make_int`(V_0) *sinon* V.

Procédure read_address. *Pas d'argument. Cette fonction est appelée dans le cas où* `#<` *a été lu. Rend un objet si on lit* `#<0x...>` *un vecteur de chiffres sinon.*

1. *Lire un caractère* c.

2. *Si c est un espace ou retour-chariot, l'ignorer. Si c est* `>`*, aller en* **4***, si c n'est ni une lettre ni un chiffre erreur.*
3. *Insérer c dans le tampon, à la position courante. S'il ne reste pas assez de place dans le tampon pour un nouveau caractère, augmenter sa taille. Repartir en* **1**.
4. *Pour chaque caractère c du tampon*
   - **4.1.** *Ne rien faire si $c_0 \leq c \leq c_9$.*
   - **4.2.** *Si $c_a \leq c \leq c_z$, incrémenter c de $c_0 - c_a$.*
   - **4.3.** *Si $c_A \leq c \leq c_Z$, incrémenter c de $c_0 - c_A$.*
   - **4.4.** *Erreur dans les autres cas.*
5. *Si les deux premiers caractères sont* `0x`*, ne pas les inclure dans le test précédent, rendre* `make_address`().
6. *Erreur si le nombre de caractères n'est pas multiple de* 8.
7. *Dans les autres cas, rendre* `make_internal`().

Procédure istring_to_number. *Argument : une chaîne de caractères s.* [Procédure utilisée pour convertir au niveau C un chaîne en entier, cas de toutes les constantes précalculées du chapitre 5].

1. *Soit t la taille de s. Si t n'est pas un multiple de 8, prendre le multiple de 8 supérieur.*
2. *Copier s dans une chaîne s' cadré à droite. Mettre le code ASCII de 0 dans les cases inoccupées de gauche.*
3. *Pour chaque caractère c de s', exécuter le point* **4** *de la procédure* `read_address`.
4. *Échanger le tampon interne et la chaîne s', appeler* `make_internal` *qui rend N, échanger à nouveau le tampon et la chaîne.*
5. *Rendre* `nx`(N).

Procédure read_complex. *Pas d'argument. Cette fonction est appelée si* `#C` *a été lu.*

1. *Appeler le lecteur Lisp pour lire un object.*
2. *Erreur si le résultat n'est pas une liste de deux objets a et b.*
3. *Erreur si a et b ne sont pas des nombres réels.*
4. *Rendre le nombre complexe $a + ib$.*

Procédure string_to_number. *Arguments une chaîne s, deux indices i et j.*

1. *Si $i < 0$, poser $i = 0$.*
2. *Si j est plus grand que le dernier indice valide dans la chaîne, remplacer j par cette valeur.*
3. *Si $j < i$ rendre 0.*
4. *Sauver i, j, et la chaîne dans des variables globales.*
5. *Rendre* `read_number`($b, 0$).

Procédure read_number. *Arguments b et s. Cette fonction est appelé par le lecteur dans le cas de* `#R` *(dans ce cas $s = 1$ et $b = 0$) ou dans le cas* `#23R` *(dans ce cas $s = 1$ et b est le nombre entre le* `#` *et la lettre* `R`*, lu en base 10). Dans le cas $s = 1$, les chiffres sont lus sur le flux d'entrée Lisp. Dans le cas contraire, dans la chaîne sauvée précédemment, à partir de l'indice i, jusqu'à l'indice j inclus. Dans le cas d'une lecture sur le flux d'entrée Lisp, si on voit un backslash, il est ignoré ainsi que tous les caractères qui suivent, jusqu'au retour-chariot inclus.*

1. *Si b n'est pas une base valide, utiliser la base courante.*

2. *Si $s = 1$, consulter le caractère qui suit. Si c'est un backslash, l'ignorer ainsi que tout ce qui suit.*

3. *Si le caractère qui suit est un slash :*

3.1. *Lire l'objet qui suit. Erreur si ce n'est ni 1 ni 2.*

3.2. *Lire l'objet N qui suit.*

3.3. *Si l'indicateur est 2, rendre N.*

3.4. *Dans les autres cas, si N est un chiffre, le rendre.*

3.5. *Si ce n'est pas un vecteur de taille 2, erreur.*

3.6. *Si le premier élément est* `t` *ou* `()`*, le signe est positif ou négatif, respectivement.*

3.7. *Si le deuxième élément est un vecteur de chiffres, $[a_n, \ldots, a_0]$, le remplacer par $\sum a_i 2^{16i}$. Erreur si ce n'est pas un chiffre. On a maintenant un nombre N'.*

3.8. *Rendre $\pm N'$, suivant le signe trouvé en* **3.6**.

4. *Consulter le prochain caractère. Si c'est un signe $\pm$, le lire.*

5. *Appeler* `read_num1`.

6. *Soit N le résultat. Lui donner le signe trouvé en* **4.**.

7. *Consulter le prochain caractère. Si ce n'est pas un slash, rendre N.*

8. *Appeler* `read_num1`.

9. *Si le résultat est D, rendre N/D.*

Macro read_num1. *Pas d'argument. Utilisée uniquement par* `read_number`*, où sont expliqués les termes « lire » et « consulter » un caractère.*

1. *Poser $n = 0$.*

2. *Consulter le caractère c qui suit.*

3. *Si* `is_a_digit`(c) *rend faux, rendre n.*

4. *Lire le caractère.*

5. *S'il est entre c_0 et c_9, lui soustraire c_0. Dans les autres cas, lui soustraire $c_a - c_0$ ou $c_A - c_0$*

6. *Remplacer n par $bn + c$, aller en* **2.**

Chapitre 5

Factorisation

Les algorithmes de ce chapitre ont été écrits pour le langage de SISYPHE. Les programmes utilisent essentiellement des entiers, petits ou grand, on ne fait plus de distinction entre les deux. On utilise également des listes et des vecteurs. On utilisera également une table de hachage ; il s'agit d'une structure qui à i associe $f(i)$. Dans SISYPHE, une table de hachage est décrite par un vecteur qui contient un certain nombre d'informations, un vecteur V et une taille n. Les informations concernent les types de i et $f(i)$. On peut par exemple imposer que i soit un entier positif ou nul, ou une liste d'entiers, on peut également accepter n'importe quel objet Lisp comme indice ; les valeurs $f(i)$ sont également typées. La valeur n est le nombre d'objets contenus dans la table. La taille du vecteur V dépend de n, si n devient trop grand, on incrémente la taille de V, si n devient trop petit, on la diminue. Pour chaque indice valide i, on calcule $j = h(i)$, sa clé de hachage ; c'est un indice dans le tableau V. À cet indice se trouve une liste d'association, i.e., une liste $(i_1, f(i_1), i_2, f(i_2), \ldots)$. La valeur $f(i)$ est obtenue en testant tous les indices dans la liste.

Pour l'algorithme de Morrison et Brillhart, on utilise une table de hachage de taille fixe $2X$, et une fonction h spéciale : si $0 \leq i < X$ alors $h(i) = i$, et sinon $h(i) = X + (i \bmod X)$. Aux indices plus petits que X, on met simplement $f(i)$. Pour factoriser $20! + 1$, X est de l'ordre de 400, et 138 valeurs avec $i > X$ sont rangées dans cette table.

Le résultat de la factorisation d'un entier est une liste d'objets de la forme (p, i) où p est un nombre premier, et i un entier non nul, le nombre à factoriser est le produit des p^i. Les facteurs sont rangés par ordre croissant, et si le nombre à factoriser est négatif, le premier terme de la liste est $(-1, 1)$.

On utilise une fonction de tri qui possède entre autres les propriétés suivantes : si les objets à trier sont des entiers, le résultat est la liste rangée dans l'ordre croissant. Si les objets sont des listes, dont le premier élément est un entier, ces entiers apparaissent par ordre croissant dans le résultat.

L'algorithme de factorisation utilise un certain nombre de constantes, qui sont calculées une fois pour toutes lors du premier appel de `ifactor` ou `isprime`. Ces constantes pourraient être calculées également lors du lancement du programme.

5.1 Fonctions simples

Supposons que n se factorise en $\prod p_i^{a_i}$, et que le nombre de facteurs est N. Le nombre de diviseurs positifs de n est $\prod(1+a_i)$, la fonction d'Euler est $\phi(n) = n\prod(1-1/p_i)$, la fonction de Möbius est $\mu(n) = (-1)^N$, si tous les exposants sont 1, la valeur est 0 sinon.

On définit aussi deux fonctions, une qui rend les diviseurs positifs de n, et une qui rend les diviseurs avec leur factorisation. Supposons que $n = \prod p_i^{a_i}$. On boucle sur les facteurs premiers de n. Supposons $n = mp^k$, et que les diviseurs de m soient $x_1, \ldots, x_q$. Alors $x_i p^j$ avec $0 \leq j \leq k$ et $1 \leq i \leq q$ est un facteur de n. Si p est un nombre premier, qui ne divise pas m, on obtient ainsi tous les diviseurs de n, et ils sont tous distincts. De plus la factorisation de $x_i p^j$ est triviale si celle de x_i est connue. La fonction qui rend les diviseurs de n peut prendre en argument la factorisation de n ; ceci est utilisé dans le cas de la factorisation des polynômes, méthode de Kronecker : on cherche les diviseurs de trois parmi quatre nombres a, b, c et d, on exclut le nombre qui a le plus de diviseurs. L'algorithme teste simplement que son argument est une liste de couples ; cela prendrait trop de temps de vérifier que les facteurs premiers sont effectivement premiers.

Algorithme 27. (Fonctions simples) *Tous les arguments et variables locales sont des entiers, sauf L, R, A, z, z' qui sont des listes.*

Fonction number-of-divisors. *Argument n.*

1. *Erreur si n n'est pas un entier > 0.*

2. *Soit $L =$* `ifactor`(n), $r = 1$.

3. *Pour chaque (p,k) dans L, poser $r = r(k+1)$.*

4. *Rendre r.*

Fonction mobius-mu. *Argument n.*

1. *Erreur si n n'est pas un entier > 0.*

2. *Soit $L =$* `ifactor`(n), $r = 1$.

3. *Pour chaque (p,k) dans L, si $k > 1$, rendre 0, sinon changer le signe de r.*

4. *Rendre r.*

Fonction euler-phi. *Argument n.*

1. *Erreur si n n'est pas un entier > 0.*

2. *Soit $L =$* `ifactor`(n), $r = n$.

3. *Pour chaque (p,k) dans L, remplacer r par $r(1-1/p)$.*

4. *Rendre r.*

Fonction divisors. *Argument L.*

0. *Initialiser R à la liste (1).*

1.1. *Si L est un entier > 0, remplacer L par* `ifactor`(L).

1.2. *Si L n'est pas une liste, erreur.*

1.3. *Si les éléments de L ne sont pas des listes de longueur 2, de la forme (a,b), où a et b sont des entiers > 0, erreur.*

2. *Tant que L n'est pas la liste vide*

2.1. *Soit (p,k) le premier élément de L. Avancer dans L.*

2.2. *Poser $A = R$, et $R = ()$.*

2.3. *Tant que A est non vide*

2.3.1. *Soit y le premier élément de A, $j = 0$, avancer dans A.*

2.3.2. *Tant que $j \leq k$, ajouter yp^j à la liste R, incrémenter j.*

3. *Trier la liste R, et rendre le résultat.*

Fonction divisors-and-fact. *Argument n.*

0. *Initialiser R à la liste $((1))$.*

1. *Si n est un entier > 0, poser $L =$* `ifactor`(L), *sinon erreur.*

2. *Tant que L n'est pas la liste vide*

2.1. *Soit (p,k) le premier élément de L. Avancer dans L.*

2.2. *Poser $A = R$, et $R = ()$.*

2.3. *Tant que A est non vide*

2.3.1. *Soit z le premier élément de A, $j = 0$, avancer dans A. Mettre dans y le premier élément de z, et dans z' le reste* [c'est la factorisation de y].

2.3.2. *Tant que $j \leq k$, si $j = 0$, ajouter z à R, sinon ajouter la liste qui contient yp^j, (y,j) et les éléments de z'. Incrémenter j.*

3. *Trier la liste R, et rendre le résultat.*

5.2 Test de primalité

Pour savoir si un nombre n est premier, on procède comme suit. Dans la liste `small_primes` se trouvent les nombres premiers plus petits que 100, à savoir 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89 et 97. Si $n < 100$, il suffit de tester que n est dans cette liste. Supposons maintenant $n > 100$. Si n a un facteur premier plus petit 100, alors n est composé. Ceci se teste en calculant le pgcd de n est du produit des nombres donnés plus haut. Si tel n'est pas le cas et si $n < 101^2$, alors n est premier. On teste de même les nombres premiers entre 100 et 1000, à savoir 101, 103, 107, ..., 991, 997 (il y a 143 nombres dans cette liste). Le nombre premier suivant est 1009, donc si $n < 1009^2$, n est premier.

Dans le cas $n > 10^6$, on a le choix entre deux méthodes: utiliser une méthode heuristique, ou une méthode exacte. La méthode exacte sera donnée dans la section suivante. La méthode heuristique est expliquée dans Knuth, section 4.5.4, algorithme P.

On suppose que $n = 1 + x2^k$. Soit a un nombre quelconque, $b_0 = a^x$, et $b_{i+1} = b_i^2$. Dans le cas où $b_0 = \pm 1$, on ne sait rien dire. Si l'un des b_i est -1, on ne sait rien dire non plus. Dans les autres cas, n est composé. La preuve est comme suit: le théorème de Fermat affirme que si n est premier, le dernier b_i est 1. S'il ne l'est pas, c'est que que n est composé. Il existe donc au moins un i avec $b_{i+1} = 1$. Prenons le plus petit. Alors $b_i^2 - 1 = 0$, et cette équation a au moins trois solutions modulo n, à savoir 1, -1, et b_i. Ceci contredit le fait que l'ensemble des entiers modulo

n est un corps, donc le fait que n est premier. Bien entendu, on a intérêt à choisir k le plus grand possible, donc x impair.

La macro `Fermat` implémente cet algorithme. Si le résultat est faux, le nombre n est composé. Dans le cas contraire on ne peut rien dire. Si on teste les 5 premiers nombres premiers, le plus petit nombre composé pour lequel la méthode rend faux est 118 670 087 467. Nous utilisons une heuristique pour trouver des facteurs potentiels de n. Si on trouve un facteur de n on le range dans la variable `ffip` (pour « factor found by isprime »). L'idée est que si on veut factoriser n, on commence par tester si n est premier, et si n est composé, on cherche ses facteurs. Dans le cas où on prouve que n est composé en exhibant un facteur, ce serait un peu dommage de l'oublier.

Une des raisons pour laquelle la méthode de Fermat fonctionne bien est la suivante : si l'algorithme P ne dit pas que n est composé pour un nombre aléatoire a, la probabilité que n est composé est au plus $1/4$. En fait, c'est beaucoup moins que cela.

Pour montrer ce théorème, écrivons $n = 1 + x2^k$ comme précédemment, factorisons $n = \prod q_i^{e_i}$, et écrivons chaque q_i comme $1 + x_i 2^{k_i}$. On peut supposer que n est impair. Les facteurs q_i sont rangés de telle sorte que $k_1 \leq k_2 \leq \cdots \leq k_s$. Soit $K = k_1$. On a $q_i = 1 \pmod{2^K}$, donc $n = 1 \pmod{2^K}$, en d'autres termes $K \leq k$. Posons $x'_i = \text{pgcd}(x, x_i)$.

La première chose à faire est de compter le nombre d'entiers b_n pour lesquels l'algorithme échoue. Il y a deux cas d'échec pour un entier q. Le premier est si $q^x = 1$. Ceci équivaut à $q^x = 1$ mod chaque $q_i^{e_i}$. Fixons un indice i. Comme nous allons le montrer plus loin, le groupe modulo $q_i^{e_i}$ est cyclique, donc a un générateur ξ. Supposons $q = \xi^r$. La condition $q^x = 1$ équivaut à $rx = 0 \pmod{B}$ où B est l'ordre du groupe, donc $(q_i - 1)q_i^{e_i - 1}$. Le nombre de solutions à $rx = 0$ est $\text{pgcd}(B, x)$. Comme q_i divise n, il est premier à x. Le pgcd est $\text{pgcd}(x, q_i - 1) = \text{pgcd}(x, x_i 2^{k_i}) = x'_i$. Le nombre de solutions à $q^x = 1$ est donc $\prod x'_i$.

Cherchons maintenant le nombre de solutions de $q^{x2^j} = -1$. La condition est maintenant $rx2^j = B/2 \pmod{B}$. Supposons d'abord $j \geq K$. Il n'y a pas de solution pour $i = 1$, car on veut $2rx2^j = (2l + 1)B$, l'exposant de 2 à droite est k_1, plus petit que l'exposant à gauche. Supposons maintenant $j < K$, donc $j < k_i$. Dans ce cas il y a au moins une solution, et le nombre des solutions est $\text{pgcd}(x2^j, B)$, ce qui est $x'_i 2^j$.

Le nombre d'entiers pour lesquels l'algorithme échoue est donc

$$b_n = (1 + \sum_{j=0}^{K-1} 2^{js}) \prod x'_i.$$

Il faut comparer cela avec n. On veut montrer $b_n/n \leq 1/4$, donc $n/b_n \geq 4$. Posons $P = b_n/n$ et $Q = 1/P$. Soit A le premier facteur de b_n. C'est aussi

$$A = 1 + \frac{2^{Ks} - 1}{2^s - 1}.$$

Supposons d'abord $s = 1$, donc $n = q_1^{e_1}$ est une puissance d'un nombre premier. On a $q_1 = 1 + x_1 2^K$, et $A = 2^K$, donc $b_n \leq q_1$. Si on suppose n composé, i.e., $e_1 > 1$, on aura $Q \geq q_1^{e_1 - 1}$. La probabilité que l'algorithme échoue dans ce cas est donc négligeable.

On considère maintenant le cas $s > 1$. On introduit $\phi(n) = \prod (q_i - 1) \prod q_i^{e_i - 1}$, et on compare $\phi(n)$ à b_n. On a $b_n \leq \phi(n)$ car seuls les nombres premiers à n peuvent satisfaire $a^y = 1 \pmod{n}$. Le quotient $\phi(n)/n$ est $\prod(1 - 1/q_i)$. Il peut être arbitrairement proche de 1. Rappelons que $q_i - 1 = x_i 2^{k_i}$, posons $r = \prod 2^{k_i}$, de telle sorte que $\phi(n) = r \prod x_i \prod q_i^{e_i - 1}$.

Soit $B = 2^{Ks}$, donc $r/B = \prod 2^{k_i - K}$ est un entier. On prétend que $B/A \geq 2^{s-1}$. En fait, si $2^s = 1/y$, il faut montrer

$$\frac{1-y^K}{1-y} + y^{K-1} \leq 2.$$

Si A_K est le membre de gauche, il est clair que $A_k = A_{k-1} + (2y-1)y^{k-2}$; la suite décroit, et la valeur 2 est obtenue pour $k = 1$. On a maintenant

$$Q = \frac{n}{b_n} \geq \frac{n}{\phi(n)} = \frac{\prod x_i}{\prod x'_i} \frac{r}{B} \frac{B}{A} \prod q_i^{e_i - 1}.$$

Chaque facteur est au moins 1, le premier et le dernier sont des entiers impairs, donc au moins 3 s'ils différents de 1. Supposons que n n'a pas de facteur premier plus petit que 1000. Le dernier facteur est donc au moins 1000 (ou 1). Ceci montre que les entiers pour lesquels $e_i > 1$ sont presque sûrement trouvés composés à la première itération. Dans la suite, on supposera donc $e_i = 1$.

On peut se restreindre au cas $s = 2$ ou $s = 3$, car $s \geq 4$ donne $B/A \geq 8$. Dans le cas $s = 3$, on a $B/A \geq 4$. Ceci montre le théorème. Nous prétendons que en général on peut mieux faire. Dans le cas $x'_i \neq x_i$ on gagne un facteur 3. On gagne un facteur 2 dans r/B si $k_i \neq K$. Le mauvais cas est donc

$$n = (1 + x_1 2^K)(1 + x_2 2^K)(1 + x_3 2^K) = 1 + x 2^k.$$

Si on développe et si on regarde les puissances de 2, on voit $K = k$. La condition $x'_1 = x_1$ entraîne que x_1 divise x. Comme $x = x_2 + x_3 + x_2 x_3 2^K \pmod{x_1}$, on a $(1 + x_2 2^K)(1 + x_3 2^K) = 1 + x_1 y_1 2^K$ pour un certain entier y_1, donc $n = (1 + x_1 2^K)(1 + x_1 y_1 2^K)$. Il y a deux relation similaires avec les indices 2 et 3. Je ne sais pas s'il existe des entiers satisfaisant ces trois conditions, et $1 + x_i 2^K$ est premier.

Le dernier cas à considérer est $s = 2$. On a

$$n = 1 + x 2^k = (1 + y 2^{k_1})(1 + z 2^{k_2}).$$

Si $k_2 = k_1 + \delta$, alors

$$\frac{r}{A} = \frac{3.2^\delta . 2^{2k}}{2 + 2^{2k}} = 2^\delta \frac{3}{1 + 2^{1-2K}} \geq 2^{\delta+1}.$$

Si $\delta \geq 2$, ceci est au moins 8. Considérons $\delta = 0$ d'abord. On a $r/A \geq 2$. Comme $k_1 = k_2$, on a $y \neq z$. Ceci implique que y et z ne peuvent diviser x (car si y divise x alors y divise z). Au moins un autre facteur 3 peut être gagné. En fait, Q est au moins 18, à moins que n ne soit de la forme $(1 + y 2^K)(1 + yz 2^K)$, où Q est voisin de $2z$.

Il nous reste le mauvais cas, où $\delta = 1$. Si $K \geq 2$, on a $r/A \geq 16/3$, et si $K = 1$ on a $r/A = 4$. Rien de plus ne peut être gagné si y et z divisent x ; cette condition entraîne $x = y = z$, et $n = (1 + 2y)(1 + 4y)$. D'après Knuth, le plus petit tel nombre est obtenu pour $y = 24969$.

Algorithme 28. (Test de primalité) *Tous les arguments et variables locales sont des entiers.*

Fonction isprime. *Argument n.*

1. *Si n n'est pas un entier, erreur.*
2. *Si $n < 100$, rendre vrai si n est dans la liste* `small_primes`*, et faux sinon.*
3. *Si le pgcd de n et du produit des éléments de* `small_primes`*, n'est pas 1, rendre faux.*

4. *Si* $n < 10201$, *rendre vrai.*
5. *Si le pgcd de* n *et du produit des nombres premiers entre 100 et 1000 n'est pas 1, rendre faux.*
6. *Si* $n < 1\,018\,081$ *rendre vrai.*
7. *Si* n *est dans la table des nombres connus comme étant premiers, rendre vrai.*
8. *Si* n *est dans la table des nombres connus comme étant composés, rendre faux.*
9. *Rendre* `isprime1`(n).

Procédure prim_decompose. *Argument* n.

1. *Poser* $x = n - 1$, $k = 0$.
2. *Tant que* x *est pair, diviser* x *par 2, incrémenter* k.
3. *Rendre* x *et* k.

Macro Fermat. *Argument* p, x, n, k.

1. *Poser* $a =$ `powermod`(p, x, n), $K = k$.
2. *Si* $a = 1$, *fin de la macro.*
3. *Tant que* $a \neq n - 1$
   - **3.1.** *Si* $K = 1$, *fin de la fonction, rendre faux.*
   - **3.2.** *Poser* $a = a^2 \bmod n$.
   - **3.3.** *Si* $a = 1$, *fin de la fonction, rendre faux.*
   - **3.4.** *Décrémenter* K.

Procédure isprime1. *Argument* n. [Dans les étapes 4 et 5, les racines carrées sont calculées via `isqrt`, si a divise n, la quantité a est positionnée dans la variable globale `ffip`].

1. *Appeler* `prim_decompose` *sur* n. *Ceci rend* x *et* k.
2. *Pour* $p = 2, 3, 5, 7, \ldots$ *appeler* `Fermat`(p, x, n, k).
3. *Si on veut un résultat exact, rendre* `isprime2`(n).
4. *Pour* $3 \leq k \leq 9$, *soit* $a = 1 + \sqrt{2n/k}$. *Si* a *divise* n, *rendre faux.*
5. *Pour* $5 \leq k \leq 20$, *poser* $a = \sqrt{n/k}$. *Si* a *divise* n, *rendre faux.*
6. *Rendre vrai.*

5.3 Vraie primalité

Soit n un entier et $G(n)$ le groupe multiplicatif des entiers modulo n premiers à n. Si n se factorise comme $\prod p_i^{r_i}$, on note par $\phi(n)$ l'ordre du groupe. C'est $n \prod(1 - 1/p_i)$. Dans le cas où n est premier c'est simplement $n - 1$. L'ordre de chaque élément du groupe divise l'ordre du groupe, en d'autres termes

$$a^{\phi(n)} = 1 \bmod n.$$

Ceci est le théorème de Fermat dans le cas général. On l'a appliqué plus haut de la façon suivante : si n est premier, alors chaque a tel que $0 < a < n$ est dans le groupe et vérifie cette relation avec

$n-1$ au lieu de $\phi(n)$. Nous allons utiliser un autre théorème, à savoir que ce groupe est cyclique dans le cas où n est premier. Ceci signifie qu'il existe un générateur d'ordre $n-1$. La réciproque est triviale : s'il existe un élément d'ordre $n-1$, le groupe a au moins $n-1$ éléments, donc aucun premier $< n$ ne peut diviser n, donc n est premier.

Soit G un groupe commutatif quelconque, et g un entier qui se factorise en $\prod q_i^{r_i}$. L'ordre d'un élément a est g si la condition $a^s = 1$ est équivalente à g divise s. Supposons qu'il existe a_i tels que $a_i^g = 1$ et $a_i^{g/q_i} \neq 1$. Alors le groupe a un élément d'ordre g. Réciproquement, si A est un élément d'ordre g, les relations précédentes sont vraies avec $a_i = A$.

Pour chaque q_i définissons $Q_i = g/q_i^{r_i}$. Posons alors $b_i = a_i^{Q_i}$ et $B = \prod b_i$. Il s'agit de montrer que B est d'ordre g. Lemme : si a et b sont d'ordre n et m, où n et m sont premiers entre eux, alors ab est d'ordre nm. Il est clair que $(ab)^{nm} = (a^n)^m (b^m)^n = 1$. Réciproquement, si $un + vm = 1$, $(ab)^s = 1$, on a $a^{sn} b^{sn} = 1$, donc $b^{sn} = 1$, donc $b^{snu} = 1$. Or $snu = s - svm$, d'où $b^s = 1$, et s est un multiple de m. De même s est un multiple de n, ce qui prouve le lemme. Il suffit de montrer que b_i est d'ordre $q_i^{r_i}$. Soit $s = q_i^{r_i - 1}$. On a $b_i^{sq_i} = 1$, et $b_i^s \neq 1$, par hypothèse.

On en déduit : si $n-1$ se factorise en $\prod q_i^{r_i}$ alors n est premier si et seulement si pour chaque q_i il existe a_i tel que

$$a_i^{n-1} = 1 \pmod{n}, \qquad a_i^{(n-1)/q_i} \neq 1 \pmod{n}. \tag{$*$}$$

La condition $b \neq 1 \pmod{n}$ peut être remplacée par $\mathrm{pgcd}(b-1, n) = 1$, car, si n est premier, ces conditions sont équivalentes, et si le pgcd est 1, ceci implique $b \neq 1$. On peut reformuler les conditions comme : n est premier si et seulement si, pour chaque facteur premier p de $n-1$, il existe a tel que

$$a^{n-1} = 1 \pmod{n}, \qquad \mathrm{pgcd}(a^{(n-1)/p} - 1, n) = 1. \tag{$**$}$$

Knuth (exercice 4.5.4.26) prétend que si $n = 1 + fr$, avec $0 < r \leq f + 1$, il suffit de tester les facteurs premiers de f. Ceci réduit le nombre de tests. Avant de montrer ce résultat, notons qu'on peut (et doit) choisir a premier. En fait, supposons $a = bc$. Écrivons $q = (n-1)/p$. On a $a^q = b^q c^q$. Si la première relation est vraie pour a, elle l'est pour b et c, la réciproque est fausse. En d'autres termes, en utilisant a plutôt que b, en perd une occasion de montrer que n est composé. Exemple : $b = c = 2$ et $n = 15$. Calculons maintenant le pgcd de la deuxième formule. On a $(b^q - 1)(c^q - 1) = a^q - 1 - (b^q - 1) - (c^q - 1)$. Si n divise $b^q - 1$ et $c^q - 1$, alors n divise aussi $a^q - 1$. Donc si n est premier et $\mathrm{pgcd}(a^q - 1, n) = 1$, pour au moins un des facteurs b ou c, $\mathrm{pgcd}(b^q - 1, n) = 1$.

La preuve du théorème est la suivante. Si P est premier et divise n, les conditions impliquent $a^{n-1} = 1 \bmod P$ et $a^{(n-1)/p} \neq 1 \bmod P$. Écrivons $f = \prod p_i^{s_i}$ et $n - 1 = Q \prod p_i^{r_i}$ où Q est premier à tous les p_i, puis $n = 1 + QR$. La discussion précédente montre qu'il existe un élément d'ordre Q dans le groupe modulo P. Ceci implique $Q \leq P - 1$, donc $P \geq f + 1 \geq r$. Supposons que n soit composé. Alors $n \geq r(f+1) = n - 1 + r$, car n a au moins deux facteurs. Cette équation donne $r = 1$, donc $f + 1 = n$, et $P \geq n$ entraîne que n est premier.

Les remarques précédentes montrent que l'on peut calculer le générateur du groupe. L'algorithme que nous allons donner calcule un générateur du groupe dans tous les cas où il est cyclique.

Montrons d'abord que le groupe $G(n)$ est cyclique si n est premier. Soit $\psi(d)$ le nombre d'éléments d'ordre d. On prétend $\psi(d) \leq \phi(d)$. Ceci est trivial s'il n'existe pas d'éléments d'ordre d. Dans les autres cas, si x est d'ordre d, alors tous les x^i ont un ordre qui divise d, et aucun autre élément ne peut avoir un ordre qui divise d. La dernière affirmation est conséquence du fait que

x^i (pour i entre 1 et $d-1$) sont d racines du polynôme $X^d-1=0$; comme on est dans un corps (n premier) ce polynôme ne peut avoir plus de d racines. La quantité x^i est d'ordre exactement d si et seulement si i est premier à d, il y a $\phi(d)$ tels exposants. Avant de conclure, notons que tout élément a un ordre qui divise $n-1$, donc la somme des $\psi(d)$ où d divide $n-1$ est $n-1$. Il est bien connu que la somme des $\phi(d)$ où d divide $n-1$ est $n-1$. La relation $0\leq\psi(d)\leq\phi(d)$ montre alors que $\psi=\phi$. En particulier $\psi(n-1)$ est non nul. Le groupe a donc au moins un générateur.

Supposons maintenant $n=p^k$ où p est un nombre premier pair (les autres cas seront traités dans la suite). Évidemment $p=2$. Le groupe $G(n)$ est d'ordre 2^{k-1}. Si $k=1$, le groupe a un élément, le générateur est 1. Si $k=2$, le groupe a 2 éléments, 1 et 3, et 3 est évidemment le générateur. Soit $m=2q+1$ un entier impair, donc un élément quelconque du groupe. On a $m^2=4q^2+4q+1$, et ceci est 1 mod 8, car $q(q+1)$ est pair. On en déduit $m^{2r}=1 \bmod 8r$, pour chaque r qui est une puissance de 2. Ceci montre que $G(n)$ n'est pas cyclique si $n>4$.

On suppose maintenant $n=p^k$ où p est un premier impair. Soit ξ un générateur modulo p. Dans le cas où ξ^{p-1} n'est pas 1 mod p^2, on pose $\theta=\xi$. Sinon on pose $\theta=\xi+p$. Développons θ^{p-1}, c'est $\xi^{p-1}+(p-1)p\xi^{p-2} \bmod p^2$. Le premier terme est 1, le second n'est pas 0, car ξ^{p-2} n'est pas zéro mod p. On prétend que θ est un générateur du groupe $G(p^k)$.

Posons $q=(p-1)p^{k-2}$. Il faut montrer que l'ordre de θ est pq. On montre d'abord $\theta^q\neq 1$, puis que l'ordre est multiple de $p-1$. Ce dernier résultat entraîne que l'ordre est $(p-1)p^a$, le premier résultat montre $a>k-2$, donc $a=k-1$, et l'ordre est pq.

Si p est un nombre premier, on note par $F(n)$ la puissance de p dans $n!$. C'est $\sum_k\lfloor n/p^k\rfloor$. Si $n=p^k$, c'est $(p^k-1)/(p-1)$. Supposons $n=ap^q$, $m=p^k-ap^q=(p^{k-q}-a)p^q$. On a $F(n)=a(p^q-1)/(p-1)+F(a)$, $F(m)=b(p^q-1)/(p-1)$ avec $a+b=p^{k-q}$. On a donc $F(n)+F(m)=(p^q-1)/(p-1)-(p^{k-q}-1)/(p-1)+F(a)+F(b)$. Soit f la puissance de p dans le nombre de combinaisons de p^k avec n. C'est $F(a+b)-F(a)-F(b)$, par ce que l'on vient de voir. Supposons a premier à p, donc b premier à p. La quantité en question est au moins $k-q$, la puissance de p dans $a+b$, car

$$\frac{(a+b)!}{a!b!}=\frac{(a+b-1)!}{a!(b-1)!}\frac{a+b}{b}$$

Le premier facteur est entier, b divise le premier facteur, car il ne peut diviser $a+b$. On en déduit : si c est le coefficient du binôme de p^{k-2} avec i, si $0<i<p^{k-2}$ et $i=ap^j$ où a est premier avec p, alors c est multiple de p^{k-2-j}.

Ceci étant dit, montrons $\theta^q\neq 1$. Écrivons $\theta^{p-1}=1+x$. Calculons θ^q modulo p^k. C'est

$$\sum_{i=0}^{p^{k-2}}\binom{p^{k-2}}{i}x^i$$

Rappelons que $x=0 \bmod p^2$, donc $x^i=0 \bmod p^{2i}$. Le terme en $i=p^{k-2}$ est nul, car la puissance de x est suffisante. Le terme en $i=0$ est non nul. Dans les autres cas, on applique le résultat précédent : dans cx^i, il y a au moins $k-2-j+2i$ comme puissance de p. Il suffit de montrer que $j+2\leq 2i$. Ceci se montre facilement, vu que $i\geq p^j$; la relation $p>2$ est utilisée ici.

On montre maintenant que l'ordre de θ est multiple de $p-1$. Si cela l'était pas le cas, il existerait un facteur premier r de $p-1$ tel que $\theta^{p^{k-1}(p-1)/r}=1$. Cette relation est modulo p^k, mais est aussi valide mod p. Écrivons $s=p^{k-1}$ et $m=(p-1)/r$. On a $\xi^{ms}=\theta^{ms}=1 \bmod p$.

Comme s est une puissance de p, premier à $\phi(p) = p - 1$, on en déduit $\xi^m = 1$, et ceci contredit le fait que ξ est un générateur mod p.

Soit maintenant n un entier quelconque, et factorisons $n = \prod p_i^{a_i}$. Le théorème des restes chinois montre que l'anneau des entiers mod n est isomorphe au produit des anneaux mod $p_i^{a_i}$. Supposons qu'il existe deux nombres premiers impairs dans la factorisation. Alors $\phi(p_i^{a_i})$ est pair. Comme l'ordre d'un produit est le ppcm des ordres, il va manquer un moins un facteur 2 dans l'ordre. Le groupe n'est pas cyclique. Le même phénomène se passe si 4 divise n, et n a un autre facteur premier. Le seul cas qui reste est le cas $n = 2p^k$ où p est premier impair. Les groupes $G(p^k)$ et $G(2p^k)$ sont isomorphes. Soit x un générateur mod p^k. Si x est pair, on ajoute p^k. Alors x est un générateur mod $2p^k$.

Algorithme 29. (Vraie primalité) *Tous les arguments et variables locales sont des entiers, sauf r, le résultat de la factorisation de n, et T une matrice de valeurs.*

Fonction generator-modp. *Argument m.* [Rend un générateur du groupe multiplicatif modulo m, ou erreur si le groupe n'est pas cyclique.]

1. *Erreur si m n'est pas un entier > 1.*
2. *Si $m = 2$, rendre 1, si $m = 3$ rendre 2, si $m = 4$ rendre 3.*
3. *Factoriser m.*
4. *Si m a plus de deux facteurs, erreur, non cyclique.*
5. *Si m a un facteur, n^e, erreur si $n = 2$.*
6. *Si m a deux facteurs $a^b n^e$, erreur si a^b n'est pas 2^1.*
7. *Poser $g =$* `generator1`(n).
8. *Si $e \neq 1$ et* `powermod`$(g, n-1, n^2) = 1$, *incrémenter g de n.*
9. *Si m est pair, et g est pair, ajouter n^e à g.*
10. *Rendre g.*

Procédure fac_aux1. *Argument n.*

1. *Factoriser $n - 1$, résultat r.*
2. *Soit s la longueur de la liste r.*
3. *Soit T une table de taille $s \times 4$.*
4. *Pour chaque (p, k) de r, et pour i l'indice dans r, poser $T[i, 0] = p$, $T[i, 1] =$* faux, *$T[i, 2] = 0$, $T[i, 3] = k$.* [Le premier i est 0.]
5. *Rendre T.*

Procédure fac_aux2. *Argument x et k* [même procédure mais $n = 1 + x2^k$].

1. *Factoriser x, résultat r.*
2. *Soit s un de plus que la longueur de la liste r.*
3. *Soit T une table de taille $s \times 4$.*
4. *Pour chaque (p, k) de r, et pour i l'indice dans r, poser $T[i, 0] = p$, $T[i, 2] = (n-1)/p$, $T[i, 3] = k$, mettre faux dans $T[i, 1]$.* [Le premier i est 1.]

5. *Poser* $T[0,0] = 2$, $T[0,2] = (n-1)/2$, $T[0,3] = k$, *mettre faux dans* $T[i,1]$.

6. *Rendre* T.

Procédure fetch_fr. *Argument* n [calcule $n = 1+fr$, pour chaque facteur p de f, calcule $(n-1)/p$ dans $T[i,2]$. Pour les autres met vrai dans $T[i,1]$.]

1. *Poser* $f = n-1$, $r = 1$.

2. *Pour chaque indice* i *de* T

2.1. *Soit* $F = T[i,0]^{T[i,3]}$.

2.2. *Soit* $r' = rF$, $f' = f/F$. *Si* $r' \leq f'+1$, *poser* $f = f'$, $r = r'$, *mettre vrai dans* $T[i,1]$.

Macro ck_aux2. *Arguments* P *et* i [vérifie le deuxième terme de ** avec $a = P$].

1. *Soit* $q = T[i,2]$.

2. *Soit* $b = \texttt{powermod}(P,q,n)$.

3. *Soit* $b = \texttt{pgcd}(n, b-1)$.

4. *Si* $b = 1$, *mettre vrai dans* $T[i,1]$.

5. *Si* $b \neq n$, *fin de la procédure,* n *n'est pas premier. Mettre* b *dans la variable globale* `ffip`.

Macro ck_aux1. *Argument* P [vérifie le premier terme de ** avec $a = P$].

1. *Poser* $i = 0$. *Tant que* $T[i,1]$ *est vrai incrémenter* i.

2. *Soit* $q = T[i,2]$.

3. *Soit* $b = \texttt{powermod}(P,q,n)$.

4. *Si* $b = 1$, *incrémenter* i, *fin de la macro.*

5. *Soit* $c = \texttt{powermod}(b, T[i,0], n)$.

6. *Si* $c \neq 1$, *fin de la procédure,* n *est composé* [Fermat].

7. *Soit* b *le pgcd de* n *et* $b-1$. *Si* $b = 1$, *mettre* $T[i,1]$ *à vrai, incrémenter* i.

8. *Dans les autres cas, mettre* b *dans* `ffip`, *et rendre faux.*

Macro ck_aux3. *Arguments* P *et* i. [On sait que n est premier ; on veut un générateur.]

1. *Tant que* $T[i,1]$ *est vrai, incrémenter* i.

2. *Soit* $p = T[i,0]$, $k = T[i,3]$, $q = (n-1)/p^k$, $r = p^{k-1}$.

3. *Soit* $b = \texttt{powermod}(P,q,n)$.

4. *Si* $\texttt{powermod}(b,r,n) \neq 1$, *multiplier* g *par* b *modulo* n, *et mettre* $T[i,1]$ *à vrai.*

Procédure generator1. *Argument* n.

1. *Poser* $g = 1$, $P = 2$.

2. *Appeler* `fac_aux1` *sur* n. *Ceci rend une table* T.

3. *Tant que pas fini :*

3.2. *Pour chaque indice* i

3.2.1. *Si* $T[i,1]$ *est vrai, ne rien faire*

3.2.2. *Sinon appeler* `ck_aux3` *sur* P *et* i.

3.3. *Si* $T[i,1]$ *est vrai pour tout* i, *rendre* g.

3.4. *Remplacer* P *par* `nextprime`(P).

Procédure isprime2. *Argument* n.

1. *Poser* $g = 1$, $P = 2$.

2. *Appeler* `fac_aux2` *sur* n. *Ceci rend une table* T. *Appeler* `fetch-fr`.

3. *Tant que pas fini :*

3.1. *Poser* $i = 0$. *Appeler* `ck_aux1`, *sauf si* P *est un nombre premier pour lequel on a testé le critère de Fermat.*

3.2. *Tant que* i *est plus petit que la taille de* T

3.2.1. *Si* $T[i,1]$ *est faux, appeler* `ck_aux2` *sur* P *et* i.

3.2.2. *Incrémenter* i.

3.3. *Si* $T[i,1]$ *est vrai pour tout* i, *rendre vrai.*

3.5. *Remplacer* P *par* `nextprime`(P).

Logarithme discret

Nous nous intéressons ici au problème suivant : soient a, b et n trois entiers ; trouver c tel que $a = b^c \bmod n$. Soit $n = p_i^{k_i}$ la factorisation de n. Il s'agit de trouver c tel que

$$\forall i \qquad a = b^c \bmod p_i^{k_i}.$$

On considère aussi un problème du même type : trouver b_i et c_i tels que

$$\forall i \qquad a = b_i^{c_i} \bmod p_i^{k_i},$$

où b_i est un générateur du groupe $G(p_i^{k_i})$.

On va d'abord chercher l'ensemble des solutions de

$$a = b^c \bmod p^k$$

dans le cas où p est un nombre premier. Soit $a = Ap^\alpha$ et $b = Bp^\beta$, où A et B sont premiers à p. Dans le cas $\alpha \geq k$, a est nul modulo p^k, et l'ensemble des c est l'ensemble des entiers $\geq k/\beta$. Celui-ci est vide si β est nul. Dans le cas $0 < \alpha < k$, il faut $c = \alpha/\beta$, mais ce n'est pas une condition suffisante. En effet, si $\alpha \neq c\beta$, la puissance de p dans $a - b^c$ est le minimum de α et $c\beta$, qui est strictement inférieur à k, et il n'y a pas de solution. Il y a donc au plus une solution. Si $\beta = 0$ ou si α/β n'est pas entier il n'y en a pas, sinon, il y a au plus une solution $c = \alpha/\beta$; il suffit de calculer $a - b^c$ modulo p^k et de conclure globalement.

Il reste à étudier le cas où a et b sont premiers à p. Supposons d'abord que p est impair. Alors le groupe modulo p^k a un générateur ξ que l'on sait calculer. Soit $a = \xi^\alpha$ et $b = \xi^\beta$. Posons $N = (p-1)p^{k-1}$, c'est l'ordre du groupe. Il s'agit de résoudre $\alpha = c\beta \bmod N$. Soit p le pgcd de α, β et N. Notons par des primes les quotients par p. Il faut résoudre $\alpha' = c\beta' \bmod N'$. Si β' et N' ne sont pas premiers entre eux, il n'y a pas de solutions. Dans le cas contraire, β' est inversible modulo N' et l'on a une solution. L'ensemble des solutions est l'ensemble des $c + iN'$ où c est une solution particulière.

Étudions maintenant le cas où b est un générateur modulo p. Dans le cas $k = 1$, on teste tous les c entre 0 et $p-2$. Dans le cas contraire, si $a = b^{c'} \bmod p^{k-1}$ on a $c = c' \bmod N$ où $N = (p-1)p^{k-2}$ est l'ordre de b modulo p^{k-1}. On teste donc tous les $c' + iN$ avec $0 \leq i < p$.

Le cas $p = 2$ est particulier, dans la mesure où le groupe n'est pas cyclique. On ne peut donc pas supposer que b est un générateur, et la méthode peut échouer. Le principe est le même, mais il faut calculer l'ordre de b modulo p^i pour chaque i. Notons que cet ordre est une fonction croissante de i, et est une puissance de 2, ce qui simplifie les calculs.

Finalement, il s'agit de trouver l'intersection d'ensembles qui sont de la forme: ensemble vide, ensemble réduit à un point, ensemble des c qui sont $\geq c_1$, ensemble des c de la forme $c = c_0 + kN$. Si l'un des ensembles est vide, il n'y a pas de solution. Si l'un des ensembles est réduit à un point c, on calcule $a - b^c$ modulo n, et on regarde si le résultat est nul. Les contraintes $c \geq c_1$ se réduisent à $c \geq \max(c_i)$. Il suffit alors de trouver l'intersection de l'ensemble des $c_0 + kN$ et des $c_1 + kM$. Si p est le pgcd de N et de M, et p divise $c_0 - c_1$, la relation de Bezout entre N/p et M/p donne une solution c_2 et l'ensemble des solutions est $c_2 + kNM/p$. Dans le cas contraire l'intersection est vide.

Algorithme 30. (Logarithme discret) *Toutes les variables locales sont des entiers, sauf L, R qui sont des listes.*

Procédure intersect. *Argument L.* [L est une liste d'objets de la forme (c, N) qui désigne l'ensemble des $c + kN$. On cherche l'intersection.]

1. *Si L est vide, rendre $(0, 1)$. Si L est réduit à un seul élément, rendre cet élément.*

2. *Soit (a, b) le premier élément de L. Avancer dans L.*

3. *Tant que L est non vide:*

3.1. *Soit (a', b') l'élément suivant de L.*

3.2. *Soit (p, u, v) la quantité* `bezout`(b, b').

3.3. *Soit $\delta = (a - a')/p$.*

3.4. *Si δ n'est pas entier, rendre la liste vide.*

3.5. *Poser $a = a + \delta ub$ et $b = bb'/p$.*

4. *Rendre (a, b).*

Procédure order2. *Arguments b, c et k.* [Trouve l'ordre de b modulo 2^k. On sait que c'est au moins c.]

1. *Poser $q = 2^k$, $b =$* `powermod`(b, c, q).

2. *Tant que $b \neq 1$, poser $b =$* `powermod`$(b, 2, q)$ *et $c = 2c$.*

3. *Rendre c.*

Procédure logd1. *Arguments p, k, a, b.* [Suppose que b est un générateur modulo p^k, rend une solution.]

1. *Poser $a' = a \bmod p$, $b' = b \bmod p$* [reste de la division par p].

2. *Poser $B = 1$. Pour $0 \leq c < p - 1$, si $B = a$, fin de la boucle. Sinon incrémenter c, multiplier B par b' modulo p. Si c devient $\geq p - 1$, rendre la liste vide.*

3. *Pour* $i = 2, 3, \ldots, k$

3.1. *Si* $i = 2$, *poser* $d = p^2$ *et* $e = p - 1$.

3.2. *Poser* $a' = a \bmod d$, $b' = b \bmod d$.

3.3. *Poser* $B = \texttt{powermod}(b', c, d)$ *et* $b' = \texttt{powermod}(b', e, d)$.

3.4. *Tant que* $B \neq a'$, *multiplier* B *par* b' *modulo* d, *ajouter* e *à* c.

3.5. *Ne pas répéter la boucle précédente plus de* p *fois. Rendre la liste vide si on n'a pas trouvé* a.

3.6. *Si* $i \neq k$, *multiplier* d *et* e *par* p.

4. *Rendre* c.

Procédure logd2. *Arguments* k, a, b. [Cas $p = 2$, rend une solution et la période.]

2. *Poser* $c = 0$, $e = 1$.

3. *Pour* $i = 2, 3, \ldots, k$

3.1. *Si* $i = 2$, *poser* $d = p^2$.

3.2. *Poser* $a' = a \bmod d$, $b' = b \bmod d$.

3.2. *Poser* $e = \texttt{order2}(b', e, i - 1)$.

3.3. *Poser* $B = \texttt{powermod}(b', c, d)$ *et* $b' = \texttt{powermod}(b', e, d)$.

3.4. *Tant que* $B \neq a'$, *multiplier* B *par* b' *modulo* d, *ajouter* e *à* c.

3.5. *Rendre la liste vide dans le cas où* c *devient plus grand que* 2^i.

3.6. *Si* $i \neq k$, *multiplier* d *par* 2 *et si* $i = k$, *poser* $e = \texttt{order2}(b, e, k)$.

4. *Rendre* (c, e).

Procédure logd3. *Arguments* p, k, a, b *et* S. [Si $S = 1$, alors b est un générateur. On suppose que p ne divise ni a ni b.]

1. *Poser* $w = (p - 1)p^k$.

2. *Si* $S = 0$

2.1. *Calculer un générateur* b' *modulo* p^k.

2.2. *Soit* $s = \texttt{logd1}(p, k, b, b')$.

2.3. *Soit* $N = \text{pgcd}(s, w)$.

2.4. *Si* $N = 1$, *poser* $S = 1$.

3. *Si* $S = 1$, *poser* $b' = b$.

4. *Soit* $c = \texttt{logd1}(p, k, a, b')$.

5. *Si* $S = 1$, *rendre* (c, w).

6. *Sinon soit* N' *le pgcd de* c *et* N. *Diviser* c, s *et* N *par* N'.

7. *Si* $\text{pgcd}(s, w) = 1$, *rendre* $(\texttt{modp}(c/s, w), w)$. *Dans le cas contraire, rendre la liste vide.*

Procédure logd4. *Arguments* p, k, a, b *et* S. [Si $S = 1$, alors b est un générateur.]

1. *Si* $S = 1$ *aller en* **3.2**.

2. *Soient* α *et* β *les puissances de* p *dans* a *et* b. [Diviser par p tant que cela est possible].

3. *Si* $\beta = 0$

3.1. *Si* α *est non nul, rendre la liste vide.*

3.2. *Calculer* `logd3`(p,k,a,b,S). *Si le résultat n'est pas la liste vide, ajouter* 1 *en tête, rendre cette liste. Dans le cas* $p=2$, *utiliser* `logd2`(k,a,b) *à la place de* `logd3`

3. *Si* $\alpha < k$, *soit* $c=\alpha/\beta$. *Si* c *est entier, rendre* $(1,c)$ *sinon la liste vide.*

4. *Si* $\alpha \geq k$, *rendre* $(2,\lceil k/\beta \rceil)$.

Fonction log-discret. *Arguments* a, b *et* n. [Si b est non nul, rend c tel que $a=b^c$ modulo n, ou la liste vide s'il n'y a pas de solution. Dans le cas contraire, rend une liste, où chaque élément est de la forme (p,k,b,c). L'entier n est le produit des p^k, les p sont premiers, b est un générateur modulo p^k et $a=b^c$ modulo p^k. On suppose n impair dans ce cas.]

1. *Si* $b=0$, *mettre* S *à 1, sinon à 0. Poser* $M=-1$, *soit* R *la liste vide.*

2. *Erreur si* a, b *et* n *ne sont pas des entiers* >0.

3. *Factoriser* n.

4. *Si* $S=1$, *erreur si* n *est pair.*

5. *Pour chaque* (p,k) *de la liste des facteurs:*

5.1. *Si* $S=1$, *soit* b *un générateur modulo* p^k.

5.2. *Appeler* `logd4`(p,k,a,b,S). *Soit* X *le résultat.*

5.3. *Si* $S=1$, *erreur si* X *est vide. Dans le cas contraire, supprimer le premier élément, y mettre* p, k *et* b *à la place.*

5.4. *Si* $S \neq 1$

5.4.1. *Si* X *est la liste vide, rendre la liste vide.*

5.4.2. *Si* X *est* $(0,c)$, *calculer* $a-b^c$ *modulo* n. *Si le résultat est 0, rendre* c, *sinon la liste vide.*

5.4.3. *Si* X *est* $(2,c)$, *remplacer* M *par le maximum entre* M *et* c. *Ne pas exécuter* **5.5** *dans ce cas.*

5.4.4. *Sinon enlever le premier élément de* X.

5.5. *Ajouter* X *au résultat partiel* R.

6. *Si* $S=1$, *rendre* R.

7. *Remplacer* R *par* `intersect`(R).

8. *Si* R *est la liste vide, rendre la liste vide.*

9. *Sinon* $R=(a,b)$. *Si* $M<0$ *rendre* a, *sinon* $a+b\lceil (M-a)/b \rceil$.

5.4 Algorithme général de factorisation

On donne ici deux fonctions qui factorisent un entier. La première ne cherche pas à trouver tous les facteurs, seulement les petits et les moyens. Elle est utilisée par exemple pour tester le critère d'Eisenstein pour un polynôme $P=\sum a_i x^i$: s'il existe un nombre premier p qui divise les a_i pour $i>0$ et p^2 ne divise pas a_m, alors P est irréductible. Si n est le pgcd des a_i pour $i>0$, on calcule une factorisation partielle de n, et on teste les premiers trouvés dans cette factorisation.

L'algorithme général de factorisation est divisé en 6 grandes parties. Nous décrirons plus loin les méthodes de Pollard, de Morrison et Brillhart, ou de Lenstra, qui sont des méthodes qui permettent d'obtenir un facteur (éventuellement composé) de n. Une des parties est la gestion de ces facteurs.

La première phase consiste à extraire les petits facteurs de n. On considère un entier Q qui est le produit des nombres premiers plus petits que 13 ou 23 (dans le texte qui suit, on a choisi 13, car Q tient sur 16bits, dans le code, on a choisi 23, et Q tient sur 32 bits). Si n est premier à Q, il n'y a pas de petits facteurs. On calcule donc le pgcd de n et Q. Notons que calculer ce pgcd est plus rapide que de tester si n est divisible par 3 et 5 (on divise n par un nombre machine, et on calcule le pgcd de deux nombres machine, ce qui est assez efficace). Les petits nombres premiers qui divisent n sont ceux qui divisent le pgcd, ce qui est donc facile à tester. Soit p un tel nombre premier. On cherche la puissance de p dans n. Pour cela, on précalcule k tel que p^k tienne sur un mot machine. On calcule le pgcd de n et p^k, et on divise n par le pgcd. Si le pgcd est $< p^k$, on a fini, ce pgcd est de la forme p^i, et on trouve i en regardant dans une table. Dans le cas contraire, on recommence. Cette façon de faire est plus efficace que de diviser n par p tant que p divise n (si l'exposant est i, il y a $i/k+1$ pgcds au lieu de $i+1$ divisions) sauf si $p=2$, car tester la parité d'un nombre est trivial.

Dans une deuxième phase, on extrait les facteurs premiers plus petits que 1700. L'algorithme est le suivant : soient P_1, P_2, P_3, P_4, P_5 et P_6 des nombres tels que le produit est égal au produit des nombres premiers plus petit que 1700. Soit P_∞ ce produit. On calcule le pgcd G de n et de P_∞. Si ce pgcd est trivial, on a fini. Sinon, pour chaque i on calcule g_i le pgcd de G et P_i, et on divise G par g_i. En d'autres termes, g_i est le produit des facteurs premiers de n qui sont des facteurs de P_i. On utilise le fait que si $g_i = n$, tous les exposants de tous les facteurs premiers de n sont 1. Dans le cas contraire, on cherche les facteurs de g_i, et pour chaque facteur p, on calcule l'exposant dans n, par divisions successives.

Factoriser g_i est simple : on sait que les facteurs premiers sont au moins a, où a est de la forme $30b+1$. Au lieu de diviser par a, $a+2$, $a+4$, etc, on divise par a, $a+6$, $a+10$, $a+12$, $a+16$, $a+18$, $a+22$, $a+28$, etc., i.e., on exclut les multiples de 2, 3 et 5. On utilise donc deux tables auxiliaires, l'une qui contient les incréments successifs de a, et l'autre les valeurs initiales de a.

Dans une troisième phase, on sait que n n'a plus de facteurs plus petits que 1700, donc si n est plus petit que le carré de ce nombre, c'est que n est premier. On teste que n est composé, via `isprime`, ce qui peut éventuellement donner un facteur de n. À partir de ce moment, on maintient une liste de (n_i, e_i), $n = \prod n_i^{e_i}$, il faut factoriser les n_i, et on les suppose premiers entre eux. Chaque fois qu'on trouve un facteur f de n_i, on rajoute un élément de plus à la liste. En fait, si $n_i = ab$, on s'arrange pour ne rajouter que des éléments premiers entre eux. L'avantage est que si a et b ne sont pas premiers entre eux, on trouve un facteur non trivial supplémentaire, par un simple calcul de pgcd. La procédure `gcd-free` extrait ces pgcd, de façon itérative : tant qu'il y a au moins une paire avec un pgcd non trivial, on rajoute un nouvel élément à la liste. Exemple : si $a = p^3 r$ et $b = p^2 q$, où r et q sont premiers entre eux, on a d'abord $(p^3 r, p^2 q)$, puis (pr, q, p^2), puis (r, q, p, p), et finalement $(r, q, 1, 1, p)$. Il faudrait en principe une table de hachage, on utilise une table de taille fixe 1000. Si cette taille ne suffit pas, c'est que n est vraiment énorme.

Pour trouver un grand facteur de n (le n_i courant) on procède comme suit : on teste d'abord que n n'est pas un carré, un cube, une puissance cinquième, puis on teste l'algorithme de Pollard, puis on teste si n n'est pas une puissance plus grande (7, 11, 13, etc). Si ceci échoue, on utilise le même algorithme que dans la deuxième phase, sauf que les P_i choisis sont les produits de tous les nombres premiers plus petits que 10^5 pour lesquels il y a des problèmes avec l'algorithme de Pollard.

Nous avons essayé cet algorithme sur un grand nombre, la factorielle de 10000. Au bout de 7 minutes de CPU sur Sparc Station 10, tous les petits facteurs on été extraits. Nous avons arrêté le programme à ce moment, car il restait un nombre de 20000 bits à factoriser, et l'appel à `isprime`

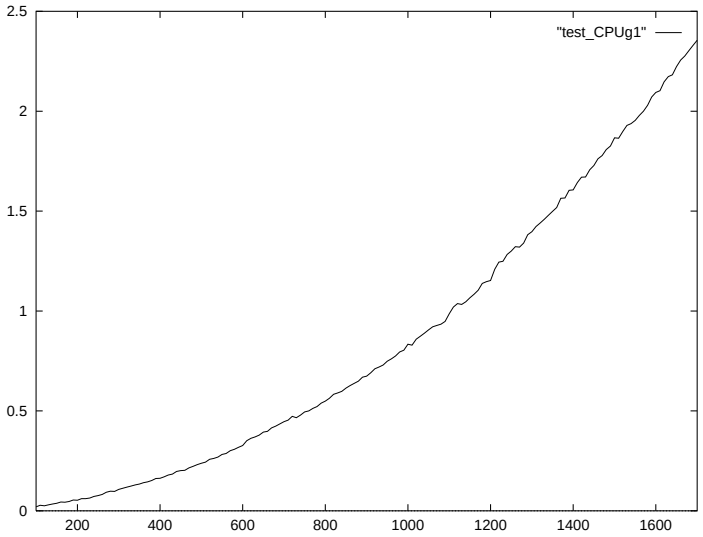


FIG. 5.1 - *Temps de factorisation de $n!$ entre 100 et 1700*

prenait trop de temps (en fait, si la machine calcule $ab \bmod n$ en une seconde, et si `powermod` exécute ceci 36000 fois, il faut de l'ordre de 10 heures de CPU pour tester qu'un nombre avec autant de chiffres est premier. Ensuite, au mieux, on coupe le nombre en deux parties égales, au pire, on extrait un petit facteur et un gros. Couper à nouveau ces nombres prend encore plusieurs heures de CPU. Nous n'avons pas fait de mesures précises, mais le calcul de $ab \bmod n$ prend plusieurs secondes, et le test de primalité doit prendre plusieurs jours).

Nous avons rajouté `factor_test`, qui est une fonction qui teste si n est divisible par tous les nombres p à partir de 1700, qui ne sont pas multiples de 2, 3 ou 5. L'algorithme s'arrête au bout de 10 échecs consécutifs. Il est donc possible de factoriser sans trop de problèmes des nombres du type factorielle.

Pour tester les divers algorithmes de factorisation, nous avons supprimé l'option précédente (étape **4.1** de la procédure `factori`). Dans les figures qui suivent nous donnons le temps de factorisation de $n!$ pour diverses valeurs de n, mesurées sur dec alpha; On a également mesuré le temps de factorisation de $n!(n+1000)!$ sur Sun, SS10/30.

À titre d'exemple, on a les résultats suivants pour la factorielle de 2500. Dans le cas où la méthode de Pollard échoue, on utilise l'algorithme de Morrison et Brillhart qui calcule b tel que $b^2 = 1$ modulo n. On montre b, et le pgcd entre $b-1$ et n. Suite à une modification du code (voir l'algorithme de Pollard), il n'y a plus d'échec dans Pollard dans la version actuelle. La boucle principale de Pollard est appelée 149 fois.

```
(extraction des petits facteurs)
On teste maintenant : 227093874678253286400409659166954650228220674432081152 82
320511487590028759632543344019659723830670506985122981733163626950281943489 86
135500330168917491141535067786790842293394003985938148715021353110639950657 57
171123047643089296266724183766923440820983064500503556665360989154131819152 35
338098438107883532913558188742236885926383026746 3
Pollard rend : 37409513223345727771657914038785842 89
```

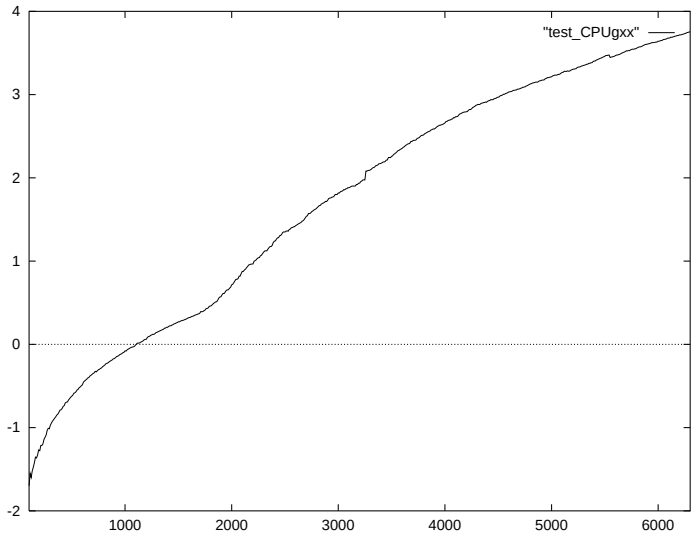


FIG. 5.2 - *Logarithme du temps de factorisation de* $n!$

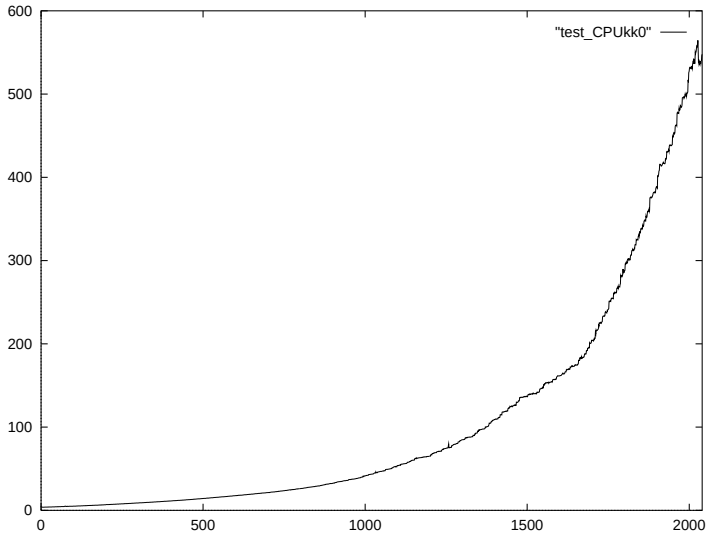


FIG. 5.3 - *Logarithme du temps de factorisation de* $n!(n+1000)!$ *entre 0 et 2000*

```
On teste maintenant : 607048462038082348024145951181344555069681914495638330316201113843200238986491702749014863414581665847836173367834140576870140477762978715366208278512387416651955999310449058856768970125165792689123689927270676269857811817350432940228828907883626335034562707141959251495456363040041362589718293409670
Pollard rend : 331082662366531162768011179

On teste maintenant : 183352537308655009262333778037065777981969673826131310937938653169949172226031140969959382718038300287934170555718041112950616820275043859158014289604871800671522074785041156061733614680192416464339512915957189808234607248017856859699379238713921330583023598941019095533173
Pollard rend : 144567158442908936146611801495375901811425291267898183

On teste maintenant : 126828623653872826017355609618978244995455109407513453103584346519449836943492656418592950719076478411656661514968383921462968720287306612990177933810925757125053853283443419852112622045531258108561270675402237176750653
Pollard rend : 260262557604339493429955383155582085 1

On teste maintenant : 48731029473199239088326129350053557713420914367913821254652649340649387613863062189947092987779379427493947182537969548611391075390255023003062693198904229061894032857796429271505768 1
Pollard rend : 43967218271028675132700932434348473268 89

On teste maintenant : 11083491607953187862268193865716957551617557234910961316689055487396061666205232888540587002526251416126311461429370300406858295616647674700112 9
Pollard rend : 34339369046444141

On teste maintenant :  32276340293156577416651831020986505818289336888198085131876839290340910107742227929166475792152130909826295431244129137152394691
Pollard rend : 30758988453283

On teste maintenant :  10493303556512641383383571853066474680810777199172785858433655250134580121123276170482031643463515758522033974014 3
Pollard rend : 39823397292479029

On teste maintenant :  26349594132930458739990455573382540738815273821960609940613741548341624899463313141542162151188 67
Pollard rend : 5653363

On teste maintenant :  46608707300292690810744782483245000787699770600190735922341695639111843515909580088068220 9
Pollard rend : 10061805529

On teste maintenant :  46322409199778016113895797084277955127729015165097945834430662289449863522510921
```

```
Pollard rend : 22695520906139

On teste maintenant :  204103749772264918208493085327063652544324627223699707941
17892418539
Pollard rend : 10160771329

On teste maintenant :  200874267477833638990354533671115602117813025751344224491
Pollard rend : 2153

On teste maintenant :  93299706213578095211497693298242267588394345448836147
Pollard rend : 3284053

On teste maintenant :  284099270668220321692426076248593635938257833 99
Pollard rend : 4138921

On teste maintenant :  68640901981028466523624412316300223159 19
Pollard rend : 2383

On teste maintenant :  28804407041975856703157537690432321 93
Pollard rend : 1823

On teste maintenant :  15800552409202334999898932402108191
Pollard rend : 1733

On teste maintenant :  91174566700532804379626797582 7
Pollard rend : 1997

On teste maintenant :  45655767000767553520093539 1
Pollard rend : 2477

On teste maintenant :  184318800972012731207483
Pollard rend : 2027

On teste maintenant :  90931820903804998129
Pollard rend : 1907

On teste maintenant :  47683178240065547
Pollard rend : 4989433

On teste maintenant :  9556833059
Pollard rend : 4330237

On teste maintenant :  4330237
Pollard rend : 2063

On teste maintenant :  4989433
Pollard rend : 2039
```

```
On teste maintenant :  4138921
Pollard rend : 1987

On teste maintenant :  3284053
Pollard rend : 1759

On teste maintenant :  10160771329
Pollard rend : 1913

On teste maintenant :  5311433
Pollard rend : 2411

On teste maintenant :  22695520906139
Pollard rend : 2389

On teste maintenant :  9500008751
Pollard rend : 2111

On teste maintenant :  4500241
Pollard rend : 1949

On teste maintenant :  10061805529
Pollard rend : 2423

On teste maintenant :  4152623
Pollard rend : 1811

On teste maintenant :  5653363
Pollard rend : 2417

On teste maintenant :  39823397292479029
Pollard rend : 2467

On teste maintenant :  16142439113287
Pollard rend : 3733643

On teste maintenant :  4323509
Pollard rend : 1931

On teste maintenant :  3733643
Pollard rend : 1789

On teste maintenant :  30758988453283
Pollard rend : 5606291

On teste maintenant :  5486513
Pollard rend : 2399
```

```
On teste maintenant :  5606291
Pollard rend : 2473

On teste maintenant :  34339369046444141
Pollard rend : 1753

On teste maintenant :  19588915599797
Pollard rend : 5247023

On teste maintenant :  3733339
Pollard rend : 2137

On teste maintenant :  5247023
Pollard rend : 2371

On teste maintenant :  439672182710286751327009324343484732688 9
Pollard rend : 4237027

On teste maintenant :  10376903019742067995483845732951 07
Pollard rend : 7638325939

On teste maintenant :  135853105806330634976113
Pollard rend : 3774647

On teste maintenant :  35990943207757079
Pollard rend : 8946430261

On teste maintenant :  4022939
Pollard rend : 2141

On teste maintenant :  8946430261
Pollard rend : 4843763

On teste maintenant :  4843763
Pollard rend : 2131

On teste maintenant :  3774647
Pollard rend : 2011

On teste maintenant :  7638325939
Pollard rend : 4469471

On teste maintenant :  4469471
Fermat: echec pour p=2, on essaie p=3.
Pollard rend : 2441

On teste maintenant :  4237027
Pollard rend : 1889
```

```
On teste maintenant :  260262557604339493429955383155582085 1
Pollard rend : 17487181818749

On teste maintenant :  148830474974130614860399
Pollard rend : 7618906583

On teste maintenant :  19534361440553
Pollard rend : 4201237

On teste maintenant :  4649669
Pollard rend : 2333

On teste maintenant :  4201237
Pollard rend : 2351

On teste maintenant :  7618906583
Pollard rend : 1721

On teste maintenant :  4427023
Pollard rend : 2237

On teste maintenant :  17487181818749
Pollard rend : 8863244713

On teste maintenant :  8863244713
Pollard rend : 1741

On teste maintenant :  5090893
Pollard rend : 2437

On teste maintenant :  1445671584429089361466118014953759018114252912678981 83
Pollard rend : 9546745693

On teste maintenant :  151430825845618276224268333503807203395069 31
Pollard rend : 44671609013192429

On teste maintenant :  338986728239176907153471839
Pollard rend : 10856445143

On teste maintenant :  31224468394034873
Pollard rend : 7889487283

On teste maintenant :  3957731
Pollard rend : 2297

On teste maintenant :  7889487283
Pollard rend : 1777
```

```
On teste maintenant :  4439779
Pollard rend : 2221

On teste maintenant :  10856445143
Pollard rend : 1933

On teste maintenant :  5616371
Pollard rend : 2393

On teste maintenant :  44671609013192429
Pollard rend : 18952740353497

On teste maintenant :  18952740353497
Pollard rend : 4682893

On teste maintenant :  4047229
Pollard rend : 2129

On teste maintenant :  4682893
Pollard rend : 2281

On teste maintenant :  9546745693
Pollard rend : 5102483

On teste maintenant :  5102483
Pollard rend : 2143

On teste maintenant :  331082662366531162768011179
Pollard rend : 35657384979482449

On teste maintenant :  9285107771
Pollard rend : 1861

On teste maintenant :  4989311
Pollard rend : 2459

On teste maintenant :  35657384979482449
Pollard rend : 2081

On teste maintenant :  17134735694129
Pollard rend : 2003

On teste maintenant :  8554536043
Pollard rend : 1873

On teste maintenant :  4567291
(echec de Pollard, appel a MB)
```

```
base= 4543880 resultat = 1951

On teste maintenant :  37409513223345727771657914038785842 89
Pollard rend : 10103222867

On teste maintenant :  37027306747370524744119 6667
Pollard rend : 2017

On teste maintenant :  183576136575956989311451
(echec de Pollard, appel a MB)
base= 160506748687761358580661 resultat =8760804211

On teste maintenant :  20954256270841
(echec de Pollard, appel a MB)
base= 10722229092500 resultat =5393413

On teste maintenant :  3885157
(echec de Pollard, appel a MB)
base= 19612 resultat 2179

On teste maintenant :  5393413
(echec de Pollard, appel a MB)
base= 5293578 resultat =2269

On teste maintenant :  8760804211
(echec de Pollard, appel a MB)
base= 8566227772, resultat= 4864411

On teste maintenant :  4864411
(echec de Pollard, appel a MB)
base= 4756362, resultat=2161

On teste maintenant :  10103222867
Pollard rend : 2069

On teste maintenant :  4883143
Pollard rend : 2113
```

Algorithme 31. (Factorisation des entiers) *Sauf mention explicite du contraire, toutes les variables locales sont des entiers. On utilise beaucoup de constantes précalculées. La liste t contient les nombres qui restent à factoriser, avec leur exposant. Dans le cas d'une factorisation partielle, B contient la quantité dont on n'a pas trouvé tous les facteurs premiers.*

Fonction ifactor. *Argument n. Rendre* `factori`$(n, 1)$.

Fonction factor-partial. *Argument n. Rendre* `factori`$(n, 0)$.

Procédure factori. *Argument n et s.*

1. *Si n n'est pas un entier, erreur.*

2. *Si* $n < 0$, *mettre* $(-1, 1)$ *dans le résultat partiel et remplacer* n *par* $-n$. *Dans les autres cas, le résultat partiel est vide.*

3. *Remplacer* n *par* `factor_small`(n).

4. *Si* $n \neq 1$, *remplacer* n *par* `factor_medium`(n).

4.1. *Si* $n \neq 1$, *remplacer* n *par* `factor_test`(n).

5. *Si* $n \neq 1$, *remplacer* n *par* `factor_big`(n, s).

6. *Trier le résultat partiel via* `sort`.

7. *Si* $n = 1$, *rendre le résultat partiel.*

8. *Dans les autres cas, mettre* n *et le résultat partiel dans deux variables globales, rendre* `FAIL`.

Procédure factor_small. *Argument* n.

1. *Soit* $E = (E_1, \ldots, E_6)$ *un vecteur contenant* $[p, k, p^2, p^3 \ldots, p^k]$ *; les valeurs de* p *sont 2, 3, 5, 7, 11 et 13.* [la table est calculée une fois pour toutes, k est choisi de telle sorte que le dernier élément soit voisin de 2^{16}. On pourrait choisir aussi 2^{32}.]

2. *Soit* g *le pgcd entre* n *et* 30030. [30030 = 2.3.5.7.11.13]

3. *Si* $g = 1$ *rendre* n.

4. *Pour* i *entre* 1 *et* 6 *faire :*

4.1. *Mettre* E_i *dans* D. *Poser* $p = D_0$, $j = D_1$, $P = D_j$.

4.2. *Si* p *ne divise pas* g, *fin de la boucle.*

4.3. *Diviser* g *par* p, *poser* $m = 0$.

4.4. *Tant que* $a = \mathrm{pgcd}(n, P)$ *n'est pas* 1 *:*

4.4.1. *Remplacer* n *par* n/a.

4.4.2. *Si* $a = p$, *incrémenter* m, *fin de la boucle.*

4.4.3. *Si* $a = P$, *incrémenter* m *de* j.

4.4.4. *Sinon décrémenter* j. *Tant que* $D_j \neq a$, *décrémenter* j. *Incrémenter* m *de* j. *Fin de la boucle.* [rappel: $D_j = p^j$.]

4.5. *Ajouter* (p, m) *au résultat partiel.*

5. *Rendre* n.

Procédure factor_medium. *Argument* n.

1. *Soit* P_{17} *le produit des nombres premiers entre 17 et 30,* P_{30} *ceux entre 30 et 60,* P_{60} *ceux entre 60 et 120,* P_{120} *ceux entre 120 et 300,* P_{300}, *ceux entre 300 et 600,* P_{600} *ceux entre 600 et 1200, et* P_{1200} *ceux entre 1200 et 1700. Soit* P_∞ *le produit de ces nombres.* [Ces nombres sont calculés une fois pour toutes].

2. *Soit* Q *la table* $[1, 31, 61, 121, 301, 601, 1201]$, $T = [6, 4, 2, 4, 2, 4, 6, 2]$, *la table des incréments* [Q_i est 1 modulo 30 ; la table T contient les différences des nombres non multiple de 2, 3 ou 5].

3. *Poser* $I = -1$, $s =$ faux.

4. *Soit* G *le pgcd entre* n *et* P_∞.

5. *Tant que* $G \neq 1$.

5.1. *Incrémenter* I.

5.2. *Mettre dans P le nombre P_k suivant, dans p le nombre Q_I, dans g le pgcd de G et P.*

5.3. *Diviser G par g.*

5.4. *Si $g = 1$ fin de la boucle.*

5.5. *Si $n = g$ mettre s à vrai, sinon à faux.*

5.6. *Si s est vrai, poser $e = 1$.*

5.7. *Si $p = 1$, poser $p = 13$, $k = 3$, sinon poser $p = p - 2$, $k = 7$.*

5.8. *Tant que $g \neq 1$.*

5.8.1. *Incrémenter p de T_k, et k de 1 modulo 8.*

5.8.2. *Diviser g par p, quotient q, reste r.*

5.8.3. *Si $r \neq 0$, ne rien faire.*

5.8.4. [ici $r = 0$] *Remplacer g par q. Si s est faux: diviser n par p tant que le résultat est entier; mettre le nombre de division dans e. Ajouter (p, e) au résultat partiel.*

6. *Si s est vrai, rendre 1, sinon n.*

Procédure factor_test. *Argument n.*

1. *Poser $k = 10$, $p = 1709$, $i = 7$. Soit T la table des incréments.*

2. *Si p divise n :*

2.1. *Poser $e = 0$. Tant que p divise n, remplacer n par n/p, incrémenter e.*

2.2. *Ajouter (p, e) au résultat partiel.*

2.3. *Si $n = 1$, rendre 1.*

2.4. *Poser $k = 11$.*

3. *Décrémenter k. Rendre n si $k < 0$.*

4. *Incrémenter p de T_i, et i de 1 modulo 8.*

5. *Repartir en* **2**.

Procédure factor_big. *Argument n et s.*

1. *Mettre dans une liste t le couple $(n, 1)$. Poser $B = 1$.*

2. *Tant que la liste t est non vide*

2.1. *Récupérer le premier couple (n, e) de la liste t. Avancer dans la liste.*

2.2. *Si $n < 2920681$, ou si* `isprime` *rend vrai, ajouter (n, e) au résultat partiel. Repartir en* **2**.

2.3. *Soit f la valeur de la variable* `ffip`. *Si f n'est pas un entier > 0, un diviseur strict de n, mettre f à 0.*

2.4. *Si $f = 0$, $n = q^2$ ou $n = q^3$ ou $n = q^5$ pour un certain q calculé via* `iroot`, *poser $f = -1$, ajouter (q, er) à t, où r est l'exposant.*

2.5. *Dans le cas où on demande une factorisation partielle, $(s = 1)$ si $f = 0$, multiplier B par Bn^e, poser $f = -1$.*

2.6.a *Si $f = 0$, poser $f =$* `pollard`$(n, 13003)$.

2.6.b *Si $f = 1$, poser $f =$* `pollard`$(n, 13004)$.

2.7. *Si $f = 1$, $n = q^r$ pour un certain q calculé via* `iroot`*, poser $f = -1$, ajouter (q, er) à t, où r est l'exposant. L'entier r est premier, au moins 7, au plus 100. Si q, partie entière de la racine potentielle, est < 1700, fin de la boucle.* [On sait que n n'a pas de facteurs plus petits que 1700].

2.8. *Si $f = 1$, poser $f = 0$.*

2.9. *Si $f = 0$*

2.9.1. *Soit $p = 4005 - 16$, $g = 1$, $A = 0$, $i = 7$, P la liste* `prpr`*, et T la table des incréments.*

2.9.2. *Soit g le pgcd de n et du premier élément de P.*

2.9.3. *Si $g = 1$, incrémenter p de 6000, avancer dans P. Si P est vide, poser $f = 0$, aller en* **2.10.***, sinon aller en* **2.9.2.**

2.9.4. *Si $g \neq n$, poser $f = g$, aller en* **2.10.**

2.9.5. *Tant que $n \neq 1$, incrémenter p de $T - i$, i de 1 modulo 8; si p divise n, ajouter (p, e) au résultat partiel, diviser n par p. Poser ensuite $f = -1$.*

2.10. *Si $f = 0$, utiliser une autre méthode pour trouver un facteur f.*

2.11. *Si $f = 0$, imprimer un message, faire comme si n était premier.*

2.11. *Si $f = -1$ ne rien faire.*

2.12. *Sinon appeler* `gcd-free` *sur n et f.*

3. *Rendre B.*

Procédure gcd-free. *Argument n et f.* [f est un facteur de n, t est le résultat partiel de la factorisation.]

1. *Poser $a = f$, $b = n/f$.*

2, *Si $a = b$, ajouter $(a, 2e)$ à t, fin de la procédure.*

3. *Sinon soit g le pgcd de a et b.*

4. *Si $g = 1$, ajouter (a, e) et (b, e) à t, fin de la procédure.*

5. *Sinon, créer deux tables de hachage T et E avec $T[1] = a/g$, $T[2] = b/g$, $T[3] = g$, $E[1] = E[2] = e$, $E[3] = 2e$. Poser $m = 3$. Mettre* trivial *à faux.*

6. *Tant que* trivial *est faux:*

6.1. *Mettre* trivial *à vrai.*

6.2. *Pour tous i, j avec $1 \leq i < j \leq m$:*

6.2.1. *Soit g le pgcd de T_i et T_j.*

6.2.2. *Si $g = 1$, prendre une autre valeur de (i, j).*

6.2.3. *Sinon, diviser T_i et T_j par g.*

6.2.4. *Incrémenter m. Mettre g dans T_m, et $E_i + E_j$ dans E_m.*

6.2.5. *Mettre* trivial *à faux. Prendre une autre valeur de (i, j).*

7. *Mettre dans t tous les (T_i, E_i) avec $T_i \neq 1$.*

5.5 Algorithme de Pollard

Les algorithmes de Pollard et ECM (courbes elliptiques) utilisent tous les deux le même principe: soit G un groupe fini (le groupe multiplicatif modulo n dans le cas de Pollard, un groupe

dépendant d'un paramètre dans le cas ECM) et a un élément de G. Si g est l'ordre du groupe, alors $a^g = 1$.

Soit p un facteur premier de n, et G_p un groupe déduit de G par un morphisme ϕ (dans le cas de Pollard, G_p est le groupe multiplicatif modulo p). Si h est l'ordre de a on a $\phi(a)^h = 1$ donc $\phi(a^h) = 1$. On espère qu'il existe au moins un p pour lequel h n'a que des petits facteurs, et donc peut être calculé. Une méthode générale du calcul de h est la suivante : on suppose que h divise un nombre R, on factorise R en $\prod p_i^{b_i}$. On fait décroitre les b_i les uns après les autres jusqu'à obtenir la bonne valeur. Dans le cas de Pollard, R est une constante, dans le cas ECM, R est pe produit des $\prod p_i^{b_i}$, pour tous les premiers $p_i < B$, et l'exposant b_i est choisit de sorte que $\prod p_i^{b_i-1} < B \leq \prod p_i^{b_i}$. Finalement, on espère $h \neq g$, donc si $b = a^h$, on a $b \neq 1$ et $\phi(b) = 1$. On espère finalement trouver un facteur non trivial de n à l'aide de cette relation.

L'algorithme $p-1$ de Pollard est décrit comme suit dans [9] « la méthode choisit un entier a premier à n, avec $a \neq \pm 1$. [Dans le code, a est de l'ordre de 13000, et n est beaucoup plus grand que cela ; la condition $a \neq \pm 1$ est trivial, a premier à n signifie que a est dans le groupe G]. L'étape 1 consiste à calculer $b = a^R \bmod n$, où R est un multiple de tous les nombres premiers $\leq B_1$. [On espère trouver un h diviseur de R]. La complexité est de l'ordre de $O(\log R) = O(B_1)$ multiplications modulo n si l'on utilise l'algorithme classique d'exponentiation. Il n'est pas nécessaire de calculer R explicitement. Si $p-1$ divise R, alors p divise $b-1$ d'après le théorème de Fermat, donc p divise $\mathrm{pgcd}(b-1, n)$. Soit $d = \mathrm{pgcd}(b-1, n)$. Si $1 < d < n$ alors d est un facteur non trivial de n, si $d = n$ alors B_1 est trop grand, on peut essayer une valeur plus petite, ou une autre valeur de a. »

Il y a deux cas d'échec possibles: $d = 1$, et $d = n$. Dans le second cas, la stratégie utilisée dépend du calcul effectif de R: dans le cas où g n'est pas un multiple de R, on peut essayer de remplacer a par une autre valeur, dans le cas contraire changer a n'arrange pas la situation. Si une factorisation de R peut être obtenue simplement, on peut factoriser R et procéder comme cela a été indiqué plus haut. Sinon on peut essayer de remplacer B_1 par B_2 plus petit, et peut-être par un B_3 entre les deux. Dans le cas ECM, rien de tout cela n'est fait, dans le cas de Pollard, on factorise R.

L'implémentation est la suivante: on suppose que $R = r_1.r_2 \ldots r_k$, où les r_i sont rangés dans une table `pollard_table`. On écrit $s_0 = a$, $s_i = s_{i-1}^{r_i}$. Alors $s_k = a^R$. La borne B_1 choisie est 2000, et R est le produit de tous les nombres premiers < 2000, les petits nombres premiers apparaissent plusieurs fois. L'algorithme échoue et rend 0 si $\mathrm{pgcd}(a^R - 1, n) = 1$. Excluons ce cas. Alors soit $d = n$, soit d est un facteur non trivial de n.

Pour chaque i l'algorithme calcule $d_i = \mathrm{pgcd}(s_i - 1, n)$ On s'arrête si $d_{i-1} = 1$ et $d_i \neq 1$. Notons que $d_0 = \mathrm{pgcd}(a-1, n)$ ne peut être n. Si ce n'est pas 1, c'est un facteur non trivial de n. Il existe donc un i satisfaisant aux conditions indiquées. Dans le cas $d_i \neq n$, on a trouvé un facteur premier non trivial de n. Supposons alors $d_i = n$. Dans ce cas, on factorise $r_1 = p_1.p_2 \cdots p_m$. Par construction, tous les facteurs sont ≤ 2000, cette factorisation est triviale (et le temps de factorisation est négligeable devant le temps de calcul de $s_{i-1}^{r_i} \bmod n$).

On applique alors l'algorithme précédent à s_{i-1} avec les nombres p_i. En d'autres termes, on écrit $S_0 = s_{i-1}$, $S_j = S_{j-1}^{p_j}$, et on calcule un pgcd à chaque itération. Par construction, il existe j tel que $\mathrm{pgcd}(S_{j-1}, n) = 1$ et $d = \mathrm{pgcd}(S_j - 1, n) \neq 1$. Le bon cas est $d \neq n$, car cela donne un facteur premier non trivial de n. Le mauvais cas est $d = n$. Dans ce cas, on recommence l'algorithme en entier avec a^{p_j} au lieu de a. L'algorithme se termine et rend 1 si cette nouvelle valeur est 1. Sinon, il est clair que si on note s_i' et S_j' les nouvelles valeurs, on aura $S_j' = 1 \pmod n$. On exécute moins d'étapes, et l'algorithme se termine un jour.

Analysons maintenant les causes d'échec de l'algorithme. L'idée essentielle est que p_j est un diviseur de h. Factorisons n en $n = \prod q_i^{b_i}$. On suppose que n est impair, et donc le groupe modulo $q_i^{b_i}$ est cyclique, engendré par ξ_i. Comme a est premier avec n, il est de la forme

$$a = \xi_i^{a_i} \bmod q_i^{b_i}. \tag{1}$$

Pour tout R, on a

- $\mathrm{pgcd}(a^R - 1, n) = 1$ si et seulement si pour tout i, $a_i R \neq 0 \pmod{q_i - 1}$.
- $\mathrm{pgcd}(a^R - 1, n) = n$ si et seulement si pour tout i, $a_i R = 0 \pmod{(q_i - 1)q_i^{b_i - 1}}$.

L'hypothèse sur R est que la première condition est fausse. Il existe donc au moins un facteur premier $q_i - 1$ de n pour lequel $a_i R = 0$ modulo q_i. L'une des continuations de l'algorithme de Pollard est la suivante : dans une table `prpr`, on range tous les nombres premiers q plus petits que 10^5 pour lesquels R n'est pas multiple de $q - 1$. Si n a un facteur premier plus petit que 10^5, soit q est dans cette liste, et on teste si q divise n, soit R est multiple de $q - 1$, et donc pour tout a, $a_i R$ est multiple de $q - 1$. L'algorithme ne peut pas rendre 0 dans ce cas.

On suppose donc

$$\exists i, a_i R = 0 \bmod q_i - 1. \tag{2}$$

La deuxième phase de l'algorithme se termine si on trouve R_0 un facteur de R, et un nombre premier p tel que

$$\forall i, a_i R_0 \neq 0 \bmod q_i - 1 \qquad a_i R_0 p = 0 \bmod (q_i - 1)q_i^{b_i - 1}. \tag{3}$$

Notons que si n a plusieurs facteurs premiers, s'il existe i tel que (2) soit fausse, l'algorithme ne peut échouer, et rend toujours un facteur de n.

Factorisons $q_i - 1$ en $q_i = 1 + \prod p_j^{\alpha_{ij}}$. Alors p est l'un des p_j. Dans le cas où p est premier à R_0, la puissance de p dans $a_i p$ est exactement α_{ij}.

On suppose que l'algorithme trouve les facteurs p_1, p_2, etc, et s'arrête avec $a = 1$. On a donc

$$a_i p_1 p_2 \cdots p_k = 0 \pmod{(q_i - 1)q_i^{b_i - 1}} \tag{4}$$

On suppose que les facteurs p_i sont tous simples dans R. Alors, pour tout i et j, la puissance de p_j dans $a_i p_j$ est exactement α_{ij}. Écrivons $q_i = 1 + u_i \prod p_j^{\alpha_{ij}}$, où les p_j sont ceux trouvés par l'algorithme, et u_i est premier à ces p_j. Soit p le produit de ces p_j. Alors $a_i p = v_i \prod p_j^{\alpha_{ij}}$. De plus, v_i est un multiple de $u_i q_i^{b_i - 1}$. Le facteur doit être premier à p.

Conclusion : il existe $\phi(p)$ valeurs différentes de a_i pour lesquelles l'algorithme échoue, parmi les $p u_i q_i^{b_i - 1}$ valeurs possibles. Cette probabilité est très faible si u_i est grand, ou si $b_i \neq 1$. L'algorithme risque donc d'échouer si tous les facteurs q_i sont simples, et les $q_i - 1$ ont presque les mêmes facteurs premiers.

Remarque : si on choisit $a = 13003$, un nombre premier, alors $a + 1 = 4.3251$. Si $a = 13004$, on a $a + 1 = 3^2.5.17^2$. Dans le premier cas $a - 1 = 2.3.11.197$, dans le second cas $a - 1$ est premier.

Remarque : nous avons constaté dans l'exemple donné plus haut un échec dans le cas (entre autres) de $n = 2161 \times 2251$. Soit q_1 le premier facteur. On a $q_1 = 1 + 2^4.5.3^3$. Un générateur de ce groupe est 271. On a $271^{123} = 13003 \bmod q_1$. Soit q_2 le premier facteur. On a $q_2 = 1 + 2.5^3.3^2$. Un générateur de ce groupe est 640. On a $640^{17} = 13003 \bmod q_2$.

Conditions: $3.41.R = 0 \bmod 80.3.9$ et $17.R = 0 \bmod 2250$, donc $R = 0 \bmod 720$ et $R = 0 \bmod 2250$. On veut une des relations vraies, l'autre fausse. Si la première est vraie, on a $R = 720k$, où k n'est pas multiple de 25. Si la deuxième est vraie on veut $R = 2250k$ où k n'est pas multiple de 8. Dans les deux cas, R doit être multiple de 90. Si $R = 90$, on a $b = a^R = 1044221$. Cet exemple montre que dans le cas où R a des facteurs multiples, il faut les prendre tous. Dans la première version, nous avions pris chaque facteur premier une seule fois. Dans la version actuelle, on a rajouté la macro `factor_rep`. Le gain de temps est appréciable dans le cas de la factorisation de 2500!, cela évite d'appeler Morrison et Brillhart sur un nombre de 24 chiffres. Le gain de temps est difficile à estimer: nous avons mis la section **2.4** après la section **2.6** (i.e., on ne teste si n est un carré, un cube, une puissance cinquième que après l'appel à pollard). D'autres modifications mineures ont également été faites. Ceci donne un gain de 30% pour le temps de factorisation de la factorielle de 3000.

Algorithme 32. (Pollard) *Tous les arguments sont des entiers, sauf les deux listes créées par les deux premières procédures.*

Procédure non-pollard. *Pas d'argument.*

1. *Soit B le double du produit des éléments de* `pollard_table`.
2. *Pour a de 4000 par pas de 6000, jusqu'à 94000*
   - **2.1.** *Mettre dans la variable Q le produit de tous les nombres premiers entre a et $a + 6000$, tels que $p - 1$ ne divise pas B.*
   - **2.2.** *Ajouter Q à la liste* `prpr`.

Procédure create-pollard-table. *Pas d'arguments.*

1. *Considérer tous les nombres premiers entre 2 et 2000, avec la multiplicité 1, sauf 2^9, 3^6, 5^4, 7 et 11 au cube, 13, 17, 19, 23, 29 et 31 au carré.*
2. *Ranger ces nombres par paquets de 6. Le premier paquet contient 2^9, le second les nombres ≤ 11. les deux paquets suivants ont 8 nombres (il reste ensuite un multiple de 6 nombres).*
3. *Choisir les paquets de telle sorte que si x_i est le produit des éléments d'un paquet, $f(x)$ est le nombre de bits de x plus le nombre de bits non nuls, la somme des $f(x_i)$ est minimale.*
4. *Remarque: $f(x)$ est le nombre de multiplications dans l'algorithme qui calcule y^x. Nous ne savons pas comment trouver le minimum, mais l'ordre utilisé par Maple est très bon, 1.2596 fois l'optimum, d'après les commentaires du fichier.*
5. *Mettre dans* `pollard_table` *les différents produits.*

Procédure pollard. *Arguments n, w.*

1. *Poser $s = w$. Soit P la liste créée par* `create-pollard-table`.
2. *Si P est vide, fin de la procédure, rendre 0.*
3. *Sinon, soit i le premier élément de P, avancer dans P.*
4. *Soit $S = s$, et $s = s^i \bmod n$.*
5. *Si $s \neq 1$*
   - **5.1.** *Soit $a = \mathrm{pgcd}(s - 1, n)$.*

5.2. *Si $a \neq 1$, rendre a.*

5.3. *Sinon repartir en 2.*

6. [Cas $s = 1$.] *Poser $s = S$.*

7. [i est un petit nombre, l'algorithme ne peut pas boucler.]

8. *Boucle sur les facteurs p de i, obtenus par* `factor_rep`(i).

8.1. *Si tous les facteurs sont épuisés, rendre 1.*

8.2. *Poser $s = s^p \bmod n$.*

8.3. *Si $s \neq 1$*

8.3.1. *Soit $a = \mathrm{pgcd}(s-1, n)$.*

8.3.2. *Si $a \neq 1$, rendre a.*

8.3.3. *Sinon repartir en 8.1.*

8.4. *Si $s = 1$, fin de la boucle.*

9. *Si i est le premier élément de la table P, rendre 1.*

10. *Remplacer w par $w^p \bmod n$, et repartir en 1.*

Macro factor_rep. *Argument n.* [Rend la liste des facteurs avec répétition de n].

1. *Soit $R =$* `ifactor`(n), *et L la liste vide.*

2. *Pour chaque élément (p, K) de R.*

2.1 *Soit $k =$* `Int_val`(K) [on espère que K est un petit entier.]

2.2. *Insérer k fois p dans la liste L.*

3. *Rendre* `nreverse`(L).

5.6 Morrison et Brillhart

Nous avons expliqué plus haut comment à partir de la décomposition en fractions continue de $\sqrt{n}$, il est possible d'obtenir des entiers b et r, notés *base* et *residue* dans le code tels que

$$b^2 = r \pmod{n}. \tag{1}$$

On utilise les variables A, B, P, Q, r et s pour ce faire, et r_n, la partie entière de la racine carrée de n. Les deux macros `MB_even_residue` et `MB_odd_residue` sont utilisées pour ce faire, le code dépendant de la parité du numéro d'itération.

L'algorithme procède comme suit : on calcule trois nombres X, B_1 et B_2. Si les nombres sont trop petits, l'algorithme peut échouer dans la mesure où on élimine trop de résidus. Si on prend les bornes trop grandes, alors l'algorithme utilise beaucoup de place mémoire. On calcule alors un certain nombre N de premiers p, et on les range dans une table H (on verra plus loin quel algorithme de rangement est choisi). Ces nombres premiers sont censés être des diviseurs de r,. On note par P_0 le produit de tous ces premiers, et P_1 est un multiple de P_0.

La procédure `MB_find_largest` prend en argument le résidu r, et le premier P_1. Le résultat est le plus grand facteur de r premier à P_1. Écrivons $r = A \prod p_i^{a_i}$ et $P_1 = \prod p_i^{b_i}$ où $b_i \geq 1$ et $a_i \geq 0$. La procédure rend A. Soit α la puissance de p_i dans r, et β la puissance dans P_1. À chaque itération, α est décrémenté de β, sauf la dernière, lorsque α devient plus petit que β. Dans le cas

où la quantité A est trop grande (plus grande que B_1) on refuse ce résidu. On le refuse également si le premier r est trop grand.

La procédure `MB_square_free` suppose que $r = A\prod p_1^{a_i}$. Elle trouve un nombre G dont le carré divise r. On remplace alors l'équation (1) par $(b/G)^2 = r/G^2$. Notons que en général G ne divise pas b, il faut donc inverser G modulo n. Le principe est le suivant: soit a le pgcd de r et de P_0. Il s'agit du produit des nombres premiers commun à r et P_0. Si g est le pgcd de a et r/a, c'est le produit des premiers qui divisent P_0 et dont le carré divise r. Alors G est le produit de tous ces g. Notons que P_0 est inversible modulo n, il en est donc de même de G.

La dernière opération est la suivante: Dans le cas $A = 1$, donc si $r = \prod p_i^{\alpha_i}$, on prend le plus grand des p_i qui divise r. On explique plus loin comment ce facteur est calculé. On notera l la quantité A, ou ce p_i. On range dans une table, à la position l, le couple (b, r).

Dans le cas où il y a déjà quelque chose, disons (b', r'), on considère $(bb')^2 = rr' \pmod n$. Par construction l est un facteur de r et r'. On peut donc simplifier. Il ne reste dans rr' que des puissance de p_i, leur exposant est au plus 2, ce qui rend la réduction sans carré peu coûteuse. Le plus grand facteur de r diminue. Avec un peu de chance, on tombe sur $r = 1$, donc

$$(b-1)(b+1) = 0 \pmod n.$$

Dans le cas où b n'est ni 1 ni -1, on a obtenu un facteur non trivial de n.

Décrivons maintenant l'algorithme utilisé pour résoudre le problème suivant: soient P_1, P_2, ..., P_N, N nombres premiers distincts rangés par ordre croissant, et r un entier produit d'un certain nombre des P_i. L'objectif est de calculer le plus grand de ces P_i.

Soit I tel que $2^{I-1} \le N < 2^I$ et $p = 2^I - 1$. Construisons un arbre binaire de taille $2p+1$. Les noeuds sont numérotés ligne par ligne, de gauche à droite, à partir de 1. Si le noeud i est sur la ligne j, on le place en position $x = (2i+1)/2^j$ et $y = j$. On a $1 < x < 2$, et chose importante, si A et B sont deux noeuds, $A < B$ équivaut à $x_A < x_B$. Ici l'ordre choisi sur les noeuds est l'ordre naturel: si C est l'ancêtre commun à A et B, alors $A < B$ si A est un descendant de gauche de C et B est un descendant de droite. On rajoute un noeud en position 0, d'indice 0, qui a un seul fils à droite, le noeud 1. Avec nos conventions, si A est un noeud d'indice i, les noeuds d'indice $2i$ et $2i+1$ sont les fils de gauche et de droite respectivement de A.

On supprime maintenant les k derniers noeuds de la dernière ligne, de telle sorte qu'il y ait $2N$ noeuds en tout. On a donc $2N = 2^{I+1} - k$, la dernière ligne compte $2^I - k$ feuilles, et l'avant dernière ligne compte $k/2$ feuilles. Ceci nous donne un total de N feuilles. On associe les valeurs P_1, P_2, etc, de gauche à droite, en commençant à la dernière ligne, et en passant à l'avant dernière si celle-ci est épuisée.

L'arbre effectivement construit ne contient que les noeuds qui ne sont pas des feuilles (le sommet a un seul fils, et si N est impair, il existe un sommet qui n'a qu'un fils de gauche). Les noeuds sont remplis comme suit: on considère chaque noeud A, en commençant par le dernier. Soit C_0 son fils de droite. Soit C_{i+1} le fils de gauche de C_i. La valeur de A est le produit des valeurs des C_i.

Notons $v(A)$ la valeur du noeud A, et $m(A)$ le plus petit facteur premier de $v(A)$. Alors 1^o $A < B$ équivaut à $m(A) < m(B)$, et 2^o $v(A)/m(A)$ est le produit de tous les $m(B)$ où B est un descendant à droite de A (on exclus à partir de maintenant la dernière ligne, elle ne sert que pour remplir le reste de l'arbre).

Preuve: par construction $v(A)$ est le produit de deux termes: un terme m qui est obtenu à partir de la dernière ligne, et v qui est le produit des termes dans l'arbre tronqué. La première

affirmation, par récurrence, montre que tous les facteurs de v/m sont plus grands que m, donc $m = m(A)$ et $v = v(A)$. Montrons ce premier point. Prenons deux noeuds A_1 et A_2 avec $A_1 < A_2$, leurs fils de droite B_1 et B_2, et leurs descendants directs vers la gauche C_1 et C_2. Il est clair que $C_1 < C_2$ (il suffit de regarder les coordonnées). Les valeurs associées sont dans le même ordre. Pour montrer le deuxième point, il faut montrer que v/m est le produit des $m(C_0)$ pour tous les descendants de C_0. Soit $Q(C_0)$ la quantité calculée. Si le fils de gauche de C_0 est C_1, on a $Q(C_0) = v(C_0)Q(C_1)$ Par récurrence, $Q(C_1)$ est le produit des $m(B)$ pour tous les descendants de C_1. Par construction, $v(C_0)$ est le produit de $m(C_0)$ et des descendants de droite de C_0. Il n'en manque aucun, ce qui achève la preuve.

L'algorithme de recherche est le suivant : soit A le noeud d'indice 1. Tant que A est dans l'arbre tronqué, soit W le pgcd de r et $v(A)$. Si $W = 1$, remplacer A par son fils de gauche, sinon remplacer r par W et A par son fils de droite. À la fin, rendre r.

On suppose que P_i est le plus grand facteur de r. Nous allons montrer que l'algorithme rend P_i. Soit A_0, A_1, ..., A_k le chemin à utiliser pour aller du noeud 0 aux noeud A_k tel que $m(A_k) = P_i$. Montrons d'abord que les A dans l'algorithme sont A_1, A_2, etc. Il n'y rien à montrer si $k = 0$. Dans le cas contraire A_1 est le sommet d'indice 1, c'est aussi le premier A dans l'algorithme. On suppose que l'algorithme est en A_j, et P_i est toujours le plus grand facteur premier de r. On suppose $j < k$. Dans un premier cas, A_{j+1} est le fils de gauche de A_j. Alors $A_k < A_j$, et tous les facteurs premiers de A_j sont plus grand que P_i. Le pgcd est trivial, et on va à gauche. Dans un second cas, A_{j+1} est le fils de droite, et A_k est un descendant à droite de A_j, donc P_i divise $m(A_j)$. Donc P_i divise W, et on va à droite. Finalement, si $j = k$, r se factorise en P_i et des facteurs plus petits, $m(A_k)$ en P_i et des facteurs plus grands, et donc $W = P_i$. À partir du moment où r est premier, la valeur de r ne bouge plus, et l'algorithme rend r. En particulier, si $k = 0$. on a $r = P_1$, et l'algorithme rend P_1.

Remarque: en faisant les essais, nous avons constaté que l'algorithme inversait modulo n des petits nombres plusieurs fois. Pour cette raison, quand on calcule b/c modulo n, si c est plus petit que 2000, on regarde si c est dans une table. S'il ne l'est pas, on calcule l'inverse modulo n, et on le range dans la table. On appelle `bezout` dans tous les cas, car nous avons constaté sur un exemple que c divisait b et n, le quotient b/c est entier, mais ensuite, la relation $b^2 = r$ n'est plus valide. Si `bezout` dit que le pgcd est non trivial, on arrête l'algorithme, car on a trouvé un facteur de n.

Exemple: on veut factoriser 20!+1. C'est un nombre qui a deux facteurs. Ce nombre se factorise en 0.4 secondes sur Sparc 10. On choisit comme constantes $X = 422$, $B_1 = 8440$ et$B_2 = 3561680$. Parmi les 661 résidus trouvés par la méthode des fractions continues, on en garde 138. Seuls 31 résidus servent. (Note: dans cette table, tous les indices sont décalés de 1, par suite d'une légère erreur de décomptage). Dans la table qui suit, on donne les résidus utiles, la factorisation, la décomposition sans carré, et dans le cas d'une réduction avec r', l'indice de r', et la nouvelle factorisation sans carré (cette table a été obtenue en traçant les résultats de `MB_find_largest` et `MB_square_free`, et en supprimant les lignes qui ne servent à rien.)

```
45)   512059880=  2^3 5 19 193 3491
      128014978   2 5 19 193 3491
47)   648343920=  2^4 3 5 7 31 59 211
       40521495        3 5 7 31 59 211
53)   240454912=  2^8 283 3319
         939277       283 3319
82)  -225623455=  5 89 97 5227 -1
```

```
133) 1391247000=  2^3 3^2 5^3 11 13 23 47
        1545830   2       5   11 13 23 47
159) 1614310391=  89 107 283 599
173) 1402544160=  2^5 3^4 5 31 3491
        1082210   2       5 31 3491
         113677   19 31 193                    REDUCTION 45
176)-1497481240=  2^3 5 23 109^2 137 -1
                  2 5 23 137 -1
190) -360899675=  5^2 31 211 2207 -1
      -14435987       31 211 2207 -1
200)-1264214601=  3^3 13 47 197 389 -1
     -140468289   3   13 47 197 389 -1
228) -643927955=  5 11 17 19 67 541 -1
270)-2130687237=  3 7^2 47 59 5227 -1
      -43483413   3     47 59 5227 -1
      359089635   3 5 47 59 89 97              REDUCTION 82
277)   59871168=  2^6 3^2 7 31 479
         103943           7 31 479
308)  -72376744=  2^3 11 61 97 139 -1
      -18094186   2   11 61 97 139 -1
315) 1080711879=  3 13 17 23 131 541
    -8227252605   3 5 11 13 19 23 67 131 -1    REDUCTION 228
323)  878409360=  2^4 3^4 5 283 479
         677785           5 283 479
         307055   5 7 31 283                   REDUCTION 277
340)   69746880=  2^6 3 5 7 97 107
        1089795       3 5 7 97 107
345)  627537393=  3^2 7 59 197 857
       69726377       7 59 197 857
372) -157897560=  2^3 3 5 41 67 479 -1
      -39474390   2   3 5 41 67 479 -1
      -17882970   2 3 5 7 31 41 67 -1           REDUCTION 277
377)  198012128=  2^5 31^2 47 137
          12878   2        47 137
          -5405   5 23 47 -1                   REDUCTION 176
           -286   2 11 13 -1                   REDUCTION 133
381) 1058673901=  19 31 67 139 193
           9313   67 139                       REDUCTION 173
       -8721658   2 11 61 67 97 -1             REDUCTION 308
  -332857581870   2 3 5 11 47 59 61 67 89 -1   REDUCTION 270
420) -171571400=  2^3 5^2 7 11 13 857 -1
       -1715714   2 7 11 13 857 -1
       -3324178   2 11 13 59 197 -1            REDUCTION 345
428)-1328804919=  3^5 13^3 19 131 -1
         -97071   3    13  19 131 -1
          84755   5 11 23 67                   REDUCTION 315
      -13505646   2 3 7 11 23 31 41 -1         REDUCTION 372
479)-1764931680=  2^5 3^4 5 23 31 191 -1
```

```
       -1361830   2 5  23 31 191 -1
485)  966419405=  5 7^2 23 41 47 89
       19722845   5  23 41 47 89
   -15007635654   2 3 11 23 41 59 61 67 -1      REDUCTION 381
      987943495   5 7 11 23 31 59 61            REDUCTION 392
486) -612960672=  2^5 3 19 23 769 -1
        -106122   2 3 19 23 769 -1
495) 1560723241=  7 11 31 197 3319
      133077637   7 11 31 197 283               REDUCTION 53
          10835   5 11 197                      REDUCTION 323
          -7670   2 5 13 59 -1                  REDUCTION 420
506) -839668599=  3^4 7 11 61 2207 -1
      -10366279       7 11 61 2207 -1
       30723077   7 11 31 61 211
         593835   3 5 11 59 61
          14973   3 7 23 31
531)   89776323   3^3 7 13 61 599
        9975147   3 7 13 61 599
    44879984877   3 7 13 61 89 107 283          REDUCTION 159
     3511558635   3 5 13 31 61 89 107           REDUCTION 323
     1485575273   7 13 31 61 89 97              REDUCTION 340
     7157709195   3 5 7 13 31 47 59 61          REDUCTION 270
         463749   3 11 13 23 47                 REDUCTION 485
             30   2 3 5                         REDUCTION 133
584) -324865125=  3^2 5^3 13 97 229 -1
       -1443845   5  13 97 229 -1
639)  494491608=  2^3 3^3 13 229 769
       13735878   2 3  13 229 769
          -68471  13 23 229 -1                  REDUCTION 486
          11155   5 23 97                       REDUCTION 584
       17028993   3 23 47 59 89                 REDUCTION 270
      -10340110   2 5 11 23 61 67 -1            REDUCTION 381
      411922203   3 7 11 23 31 41 61            REDUCTION 372
          36285   3 5 41 59                     REDUCTION 485
          -3198   2 3 13 41 -1                  REDUCTION 495
         713713   7 11 13 23 31                 REDUCTION 428
            429   3 11 13                       REDUCTION 506
             -6   2 3 -1                        REDUCTION 377
662) -140499409=  31 61 191 389 -1
   130422820281   3 13 31 47 61 191 197         REDUCTION 200
   -66102624258   2 3 11 31 47 59 61 191 -1     REDUCTION 420
      641935635   3 5 11 23 47 59 61            REDUCTION 479
          30597   3 7 31 47                     REDUCTION 485
       21411390   2 3 5 7 11 13 23 31           REDUCTION 133
           1430   2 5 11 13                     REDUCTION 506
             -5   5 -1                          REDUCTION 377
             -6   2 3 -1                        REDUCTION 331
              1                                 REDUCTION 639
```

Algorithme 33. (Morrison et Brillhart) *Toutes les variables et paramètres sont des entiers. On utilise une table auxiliaire P, un vecteur H, et une table de hachage* `sq`.

Macro MB_even_residue. *Pas d'arguments.*

1. *Diviser $2r_n - s$ par P, quotient q, reste r.*
2. *Mettre dans A le reste de $qB + A$ par n.*
3. *Remplacer Q par $Q + q(r - s)$.*
4. *Mettre A dans* base *et $-Q$ dans* residue.

Macro MB_odd_residue. *Pas d'arguments.*

1. *Diviser $2r_n - r$ par Q, quotient q, reste r.*
2. *Mettre dans B le reste de $qA + B$ par n.*
3. *Remplacer P par $P + q(s - r)$.*
4. *Mettre B dans* base *et P dans* residue.

Macro MB_check_loop. *Pas d'argument. Cette macro utilise des variables locales c_i, c_j, c_q, c_A et c_s.*

1. *Au premier appel, poser $c_i = c_j = 1$, $c_A = c_q = c_s = 0$.*
2. *Si $c_q = q$, $c_A = A$ et $c_s = s$, rendre 0, ça boucle.*
3. *Si $c_i = c_j$, remplacer c_i par $2c_i$, c_j par 1, c_q par q, c_A par A et c_s par s.*
4. *Dans les autres cas, incrémenter c_j.*

Procédure MB_find_largest. *Argument B_0, B_1, P_1, et r.*

1. *Soit r la valeur absolue de r.*
2. *Soit a le pgcd de r et P_1.*
3. *Tant que $a \neq 1$*
   - **3.1.** *Diviser r par a.*
   - **3.2.** *Si $r > B_2$, rendre 0.*
   - **3.3.** *Remplacer a par le pgcd de r et a.*
4. *Si $r > B_1$, rendre 0.*
5. *Sinon rendre r.*

Macro MB_invert. *Argument b, G et n.*

1. *Si* `IV_table` *n'est pas initialisée, allouer un vecteur de taille 2000, le remplir avec* faux.

2.1. *Si G n'est pas un petit entier entre 1 et 1999, poser $L =$* `bezout`(G, n). *Si le premier élément de la liste n'est pas 1, le rendre* [c'est le pgcd, donc un facteur non trivial de n], *sinon remplacer L par le deuxième élément de la liste.*

2.2 *Si G est un petit entier, et* `IV_table` *ne contient pas* faux *en position G, mettre cette quantité dans L.*

2.3. *Sinon, calculer L comme en* **2.1**, *mettre L dans la table en position G.*

3. *Remplacer* b *par* $bL \pmod n$.

Procédure MB_square_free. *Arguments* base, residue *et* P_0.

1. *Soit* a *le pgcd entre* residue *et* P_0.

2. *Soit* g *le pgcd entre* a *et le quotient de* residue *et* a.

3. *Tant que* $g \neq 1$, *diviser* residue *par* g^2, *multiplier* G *par* g, *mettre dans* a *le pgcd entre* residue *et* P_0, *refaire* **2**.

4. *Diviser* base *par* G *modulo* n *via* `MB_invert`.

5. *Rendre les nouvelles valeurs de* base *et* residue.

Macro MB_update. *Arguments* base, residue, *et* l.

1. *Si* $l \neq 1$, *aller en* **4**.

2. *Soit* r *la valeur de* residue. *Si* $r = -1$, *poser* $l = -1$, *aller en* **4**, *sinon remplacer* r *par sa valeur absolue.*

3. *Poser* $i = 1$. *Tant que* $i < N$ *:*

3.1. *Soit* W *le pgcd entre* r *et* H_i.

3.2. *Si* $W = 1$, *poser* $i = 2i$, *sinon* $r = W$ *et* $i = 2i + 1$.

4. *Si le tableau* `sq` *ne contient rien en position* l, *y mettre la liste* (base, residue). *Poser* $l = 0$. *Fin de la macro.*

5. *Remplacer* base *par le produit avec le premier terme de la liste. Diviser* base *par* l *modulo* n *via* `MB_invert`.

6. *Remplacer* residue *par le produit avec le second terme. Diviser par* l^2.

7. *Poser* $l = 1$.

Macro MB_init. *Pas d'argument.*

1. *Créer deux tables de hachage* `sq` *et* H, *et un vecteur* P *de taille 1000.*

2. *Soit* $X = \sqrt{l(n) * 599/4} - 11$, $l(n)$ *nombre de chiffres de* n *en base 10.*

3. *Poser* $X = \sqrt{\sqrt{\sqrt{\sqrt{10^X}}}}$.

4. *Soit* B_1 *le minimum de* X^2 *et* $20X$, $B_2 = XB_1$.

5. *Remplir la table* P *avec 2, et tous les nombres premier* p *tels que* $t = n^{(p-1)/2} = 1 \bmod p$. *Ne prendre que ceux qui sont* $< X$. *Dans le cas* $t = 0$, *rendre* p. *Erreur si la table* P *déborde. Soit* N *le nombre de premiers calculés.*

6. *Poser* $p = 1$. *Tant que* $p \leq N$ *remplacer* p *par* $2p$. *Décrémenter* p.

7. *Poser* $i = N - 1$. *Tant que* $i \geq 0$

8.1. *Soit* $W = 1$, $j = 2i + 1$.

8.2. *Tant que* $j < N$, *poser* $W = WH_j$, $j = 2j$.

8.3. *Si* $j > p$, *soit* $k = j - p$ *sinon* $k = N + j - p$.

8.4. *Mettre* WP_k *dans* H_i.

8.5. *Décrémenter* i.

8. *Poser* $P_0 = H_0$ *et* $P_1 = 1440 P_0$.

Procédure MB. *Argument* n.

1. *Poser* base $= 0$, residue $= 0$ $r_n =$ `isqrt`(n), $A = 0$, $Q = n$, $s = 0$, $q = 0$, $r = 0$.

2. *Poser* $P = 1$, $B = 2$.

3. *Appeler* `MB_init`.

4. *Tant que* `MB_check_loop` *ne fait pas sortir, et qu'aucun facteur n'a été trouvé:*

4.1. *Aux itérations paires appeler* `MB_even_residue`, *aux autres* `MB_odd_residue`.

4.2. *Aux itérations impaires appeler* `MB_check_loop`.

4.3. *Appeler* `MB_find_largest` *sur* B_0, B_1, P_1 *et* residue. *Ceci rend* l.

4.4. *Tant que* l *est non nul*

4.4.1. *Appeler* `MB_square_free` *sur* base, residue *et* P_0.

4.4.2. *Si* residue *est 1 : si* base *n'est ni 1 in* n, *rendre le pgcd entre* n *et* base $- 1$, *sinon poser* $l = 0$.

4.4.3. [residue n'est pas 1] *Appeler* `MB_update` *sur* base, residue *et* l.

5.7 Lenstra

L'algorithme que nous allons présenter ici possède quelques ressemblances avec l'algorithme de Pollard. On considère un groupe fini, un élément x du groupe, et un entier N tel que $x^N = 1$. L'ordre du groupe n'est pas connu a priori, on va donc choisir N très grand. Si p est un facteur premier de n, on regarde modulo p. L'idée est de trouver un N pour lequel $x^N = 1$ modulo un seul p, ce qui nous donne p, donc un facteur non trivial de n.

Un théorème de géométrie

Commençons par un peu de géométrie. Soient F_1 et F_2 deux cubiques dans le plan projectif complexe. Les points sur la cubique sont les racines de

$$Ax^3 + Bx^2y + Cxy^2 + Dy^3 + Ex^2z + Fxyz + Gy^2z + Hxz^2 + Iyz^2 + Jz^3 = 0 \qquad (*)$$

où (x, y, z) et $(\lambda x, \lambda y, \lambda z)$ sont considérés comme étant les mêmes points pour tout λ non nul. Le théorème de Bezout dit que deux cubiques s'intersectent en 9 points, comptés avec leur multiplicité, sauf si les cubiques ont une composante en commun, cas où le nombre de points d'intersection est infini.

On supposera ici que F_1 est un polynôme irréductible, et qu'aucun point d'intersection avec F_2 n'est double, et les cubiques distinctes. Il y a donc exactement 9 points d'intersection. On considère alors une troisième cubique F, qui passe par 8 de ces 9 points. Théorème: F passe aussi par le dernier point.

On va montrer un petit lemme: soit F une cubique quelconque, P_4, P_5, P_6, P_7 et P_8 cinq points distincts, tels que 4 d'entre eux ne soient pas alignés. Il existe alors une unique conique C qui passe par ces cinq points. Soit de plus P_1, P_2, A et B quatre points alignés, non sur C. Si ces 9 points sont sur F, alors $F = 0$ est formé par la réunion de la droite et de la conique. La preuve

est la suivante : par changement de coordonnées, on peut supposer que la droite a pour équation $x = 0$. Faisons $x = 0$ dans l'équation (*). Il reste une équation homogène de degré 3 en y et z qui a 4 racines, donc les 4 coefficients sont nuls, et donc $x = 0$ est en facteur dans F. Le second facteur de F est un polynôme de degré 2, disons une courbe C. Le seul point à vérifier est que C est l'unique conique qui passe par ces points. Ceci se vérifie facilement dans une base.

La première partie du lemme montre que parmi les 8 points d'intersection considérés, il n'y en a pas 4 d'alignés (le polynôme F_1 est irréductible). Donc, pour tout choix de 5 points, il passe une unique conique. Soient P_4 à P_8 ces cinq points. Je prétend qu'il existe au moins deux points, disons P_1 et P_2, qui ne sont pas sur cette conique. La raison en est que si 7 points de $F = 0$ sont sur une conique C, alors C est un facteur de F. La preuve est un peu plus délicate que dans le cas d'une droite, mais est une conséquence du théorème de Bezout : une courbe de degré 2 et une courbe de degré 3 ont 6 points d'intersection, sauf s'il y en a une infinité.

Étant donné notre choix de P_1, P_2, P_4 à P_8, notons L la droite qui passe par P_1 et P_2, et par C la conique qui passe par les 5 autres points. Supposons que P_3 ne soit ni sur L ni sur C. On choisit alors A et B sur L, non sur C, distincts de P_1 et P_2. Si P_3 est sur L, non sur C, on choisit A sur L, B ni sur L, ni sur C. On est ramené au cas précédent, en échangeant B et P_3. Dans le cas où P_3 est sur C, on choisit A sur C, et B ni sur L, ni sur C. Affirmation : aucune cubique ne passe par ces 10 points. Il y a deux cas à tester : dans le cas où 4 des points sont alignés, on sait que la cubique est la réunion de L et C. Donc P_3 devrait être sur cette réunion, alors que par construction, il ne l'est pas. Dans le deuxième cas, il y a 7 points sur une conique, la cubique est la réunion de L et C et le résultat est le même.

Ayant fixé les points A et B considérons $F^* = aF - bF_1 - cF_2$. Le système $F^*(A) = F^*(B) = 0$ a au moins une solution non nulle en (a, b, c). On trouve donc F^* qui passe par les dix points. Ceci contredit ce qu'on vient de montrer plus haut. Donc F^* est identiquement nul. Le cas $a = 0$ est exclus, cela signifie que F_1 et F_2 sont proportionnelles. On en déduit que si $F_1(P_9) = F_2(P_9) = 0$ alors $F(P_9) = 0$, et c'est ce que l'on voulait montrer.

Courbes elliptiques

Soit maintenant K un corps quelconque, A, B, C et D des éléments de K. On considère l'ensemble E des points P de K^2 qui ont des coordonnées homogènes (x, y, z) satisfaisant

$$Azy^2 = x^3 + Bzx^2 + Cz^2x + Dz^3. \tag{0}$$

Dans le cas $z = 0$, on a $x = 0$, donc $y = 1$. Ce point à l'infini sera noté O dans la suite. Pour tous les autres points, on pourra supposer $z = 1$, et considérer l'équation

$$Ay^2 = x^3 + Bx^2 + Cx + D. \tag{1}$$

Si P est un point de coordonnées (x, y) on notera par $\overline{P}$ le point de coordonnées $(x, -y)$. Si P est sur la courbe, il en sera de même de ce nouveau point. On supposera que l'équation $y = 0$ n'a que des racines simples en x. Ceci signifie que la tangente à la courbe est définie en chaque point, et que la courbe n'a pas de point double. Notons que le polynôme qui définit la courbe est irréductible. Étant donnés deux points P_1 et P_2, notons par P_1P_2 le troisième point d'intersection de la droite passant par P_1 et P_2 avec la courbe (ou la tangente si les deux points sont confondus). Si ce point est Q, alors $\overline{Q}$ sera appelé la somme des deux points et noté $P_1 + P_2$.

Supposons $P_3 = P_1 + P_2$, et que les coordonnées de P_i sont x_i and y_i. On a alors les équations :

$$\delta = \frac{3x_1^2 + 2Bx_1 + C}{2Ay_1} \text{ si } P_1 = P_2 \qquad \delta = \frac{y_1 - y_2}{x_1 - x_2} \text{ sinon.} \tag{2}$$

$$y_2 + y_3 = \delta(x_2 - x_3) \qquad x_1 + x_2 + x_3 + B = A\delta^2. \tag{3}$$

Preuve : il est clair que $O + P = P + O = P$, et $P + \overline{P} = O$. On peut donc supposer qu'aucun point n'est à l'infini.

Supposons d'abord que les deux points sont distincts. Écrivons $x_3 = x_2 + \lambda(x_1 - x_2)$ et $-y_3 = y_2 + \lambda(y_1 - y_2)$, la condition d'alignement. Éliminons λ entre ces deux équations. On obtient $y_2 + y_3 = \delta(x_2 - x_3)$. Dans le cas où les deux points sont confondus, il faut remplacer ces équations par $x_3 = x_2 + \bar{x}$ et $-y_3 = y_2 + \bar{y}$ où le quotient de $\bar{x}$ et $\bar{y}$ est la pente de la tangente. Le résultat est le même.

On va maintenant dire que le point somme est sur la courbe. On considère la différence $Ay_2^2 - Ay_3^2 = A(y_2 - y_3)(y_2 + y_3)$. On va l'écrire en fonction de x_2 et x_3, et introduire δ. Ceci donne

$$A\delta(y_2 - y_3) = x_2^3 + x_2x_3 + x_3^2 + Bx_2 + Bx_3 + C. \tag{$*$}$$

On a la même équation avec x_1 au lieu de x_2. Faisons la différence, et mettons en facteur la quantité $\delta = (y_2 - y_1)/(x_2 - x_1)$. Ceci donne la seconde équation de (3) dans le cas où les points ne sont pas confondus. S'ils le sont, on obtient le même résultat en faisant la différence entre l'équation $(*)$ et $2Ay_2\delta = 3x_2^2 + 2Bx_2 + C$.

On va maintenant montrer que $+$ définit une opération de groupe commutatif sur E. Le seul point non trivial à montrer est l'associativité. Il s'agit de montrer que $X = Y$ si $X = P_1(P_2 + P_3)$ et $Y = (P_1 + P_2)P_3$. Il s'agit de montrer une grosse identité algébrique à coefficients entiers qui dépend des paramètres A, B, C et D. Il suffit de la montrer sur les nombres complexes. Soit C_1 la cubique formée des trois droites L_1, L_2 et L_3, où L_1 passe P_1, P_2 et P_1P_2, L_2 passe par P_3, $P_1 + P_2$ et $P_3(P_1 + P_2)$ et L_3 passe par P_2P_3, $P_2 + P_3$ et O. Ces 9 points sont sur E. Soit C_2 la cubique formée des trois droites l_1, l_2 et l_3, où l_1 passe par P_3, P_2 et P_3P_2, l_2 passe par P_1, $P_2 + P_3$ et $P_1(P_2 + P_3)$ et l_3 passe par P_2P_1, $P_2 + P_1$ et O. Ces neuf points sont aussi sur E.

Appliquons maintenant le théorème. Les cubiques C_1 et E ont 9 points d'intersection, à savoir P_1, P_2, P_3, P_1P_2, $P_1 + P_2$, P_2P_3, $P_2 + P_3$, O, et un neuvième point Y. La cubique C_2 passe par les 8 premiers, et par le lemme, passe aussi par le dernier Y. Or, l'intersection de C_2 et de E est formée de 9 points, les 8 points cités précédemment, et le point X. On en déduit $X = Y$. Le lecteur intéressé montrera que ce résultat est également valide si certains des points sont confondus.

Calcul de NP

Nous verrons plus loin l'intérêt du calcul de NP où N est un entier et P un point. Pour l'instant, nous allons donner des formules pour calculer ceci. Supposons que les coordonnées de P sont x et y, celles de $2P$ sont x' et y'. Alors

$$x' = \frac{(x^2 - C)^2 - 4D(2x + B)}{4(x^3 + Bx^2 + Cx + D)}. \tag{5}$$

La preuve est simple : on a $4A^2y^2\delta^2 = (3x^2 + 2Bx + C)^2$. Remplaçons $A\delta^2$ par (3), Ay^2 par (1) et développons. Supposons maintenant que $P + Q$ est la quantité à calculer, connaissant $P - Q$. Si

x_0, x_3 sont les coordonnées en x de $P \pm Q$ et x_1, x_2 celles de P et Q, alors

$$x_0 x_3 = \frac{(x_1 x_2 - C)^2 - 2(B + x_1 + x_2)D}{(x_1 - x_2)^2}. \tag{6}$$

La preuve est simple : soit $\delta = (y_1 - y_2)/(x_1 - x_2)$ et $\delta' = (y_1 + y_2)/(x_1 - x_2)$. On sait $x_1 + x_2 + x_3 + B = A\delta^2$, $x_1 + x_2 + x_0 + B = A\delta'^2$. Calculons $x_0 x_3$. On a besoin des quantités $\delta^2 + \delta'^2$ et $\delta^2 \delta'^2$, donc $y_1^2 + y_2^2$ et $(y_1^2 - y_2^2)^2$ qui sont obtenues facilement à partir de (1). Il suffit de développer.

Soit P un point et N un entier. On veut calculer NP. En général, on utilise une méthode binaire, la même que pour les calculs de puissance. Dans notre cas, on voudrait utiliser les relations (5) et (6), dans la mesure où elles ne dépendent pas de y. Si on regarde les bits de N en commençant par le bit de poids fort (en général on fait le contraire), on voit qu'il faut calculer $2mP$ ou $(2m+1)P$ à partir de P et mP. On suppose que $(m+1)P$ est également connu. Il faudra donc également calculer $2(m+1)P$. Calculer le double d'une quantité est facile. On calcule $(2m+1)P$ en disant que c'est la somme de mP et $(m+1)P$, sachant que la différence est P, et donc on peut utiliser 6.

On utilise l'algorithme qui suit : Soit $N_0 = N$. On suppose que a, b et c sont trois entiers tels que

$$\pm c = a - b, \qquad aN + bM = N_0.$$

Cette équation est $(a+b)N + b(M-N) = N_0$ ou $a(N-M) + (a+b)M = N_0$. Il faut calculer $a+b$, mais on connait a, b et $a-b$. On jette l'un des deux a, b, en fait ce sera la nouvelle valeur de c. On choisit l'un ou l'autre des transformations de sorte que $N \geq 0$ et $M \geq 0$. On choisit $a = 1$, et $b = 0$ comme condition initiale. La quantité M est donc quelconque. Il est clair qu'il vaut mieux la prendre $< N$. En fait, le rapport optimal est le nombre d'or (M et N sont deux nombres de Fibonacci successifs). On supposera que N est premier. Alors N et M restent premiers entre eux, et l'algorithme se termine si $M = 0$, donc $N = 1$, donc $a = N_0$.

On calcule aP, bP et cP, et non a, b, c. À la première itération, il faut calculer $a+b$, mais comme $b = 0$, il n'y a rien à faire. Les nouvelles valeurs de a et b sont donc 1 et 1. À la deuxième itération, il faut calculer $a + b$, donc $2P$. C'est le seul cas où il faut doubler un point. Le cas embêtant est si $aP = bP$ avec $a \neq b$. Ceci signifie $kP = 0$, avec $k < N_0$. Or le but du jeu est justement de trouver un tel k. Ceci sera expliqué plus loin. Remarque : au lieu de calculer NP, on calcule $N^k P$ où N^k est grand, disons de l'ordre de 10^6. Ceci est fait par la fonction `elliptic_power`.

Groupe elliptique modulo p

Soit p un nombre premier. On note F_p le corps des entiers modulo p. Fixons A, B, C et D. Ceci nous donne une courbe E, munie d'une structure de groupe. L'ordre L de ce groupe (le nombre d'éléments) est assez facile à calculer. Pour chaque entier i, soit y_i la quantité $(i^3 + Bi^2 + Ci + D)/A$. Si cette quantité est 0, il existe un point unique sur E dont la première coordonnée est i. Si cette quantité est un carré, il en existe 2, sinon aucun. Rappelons que $\left(\frac{a}{p}\right)$ est le symbole de Legendre, il vaut 0 si a est 0, 1 si a est un carré modulo p, et -1 sinon. Si on n'oublie pas le point à l'infini, l'ordre sera

$$L = 1 + \sum_{i=0}^{p-1} \left[\left(\frac{y_i}{p}\right) + 1\right]. \tag{7}$$

Cette relation n'est pas exploitable. Par contre, on peut montrer que

$$p + 1 - 2\sqrt{p} \leq L \leq p + 1 + 2\sqrt{p}. \tag{8}$$

Tout élément de cet intervalle peut être un ordre, et les ordres sont en général bien répartis. En fait, nous allons choisir la courbe de façon aléatoire. L'idée est de calculer LP pour un point P. Le résultat sera le point à l'infini. Donc le dénominateur de la première coordonnée sera 0 modulo p. Soit maintenant p un facteur de n. Avec un peu de chance, ce dénominateur ne sera pas multiple de n, et donc le pgcd avec n donnera un facteur de n. Pour éviter de calculer des inverses modulo n, on sépare numérateur et dénominateur. Bien entendu, on ne calcule pas LP, mais on multiplie P par une puissance de 2, de 3, de 5, etc. On espère multiplier P par un facteur de L.

Choix des courbes

Au lieu de choisir une courbe elliptique quelconque, $Ay^2 = x^3 + Bx^2 + Cx + D$ on peut par changement de variables annuler B ou D, et positionner A ou C à 1. Certaines personnes utilisent $y^2 = x^3 + ax + b$. On préfère $Ay^2 = x^3 + Bx^2 + x$. La raison principale est que les équations (5) et (6) se simplifient beaucoup.

Supposons $B' = (B+2)/4$, $x = a/b$ est rationnel dans (5). Alors x' est

$$x' = \frac{(a^2 - b^2)^2}{4ab[(a-b)^2 + 4abB']}.$$

Ceci peut se calculer avec une multiplication de moins :

$$t_1 = (a+b)^2 \qquad t_2 = (a-b)^2 \qquad x' = \frac{t_1 t_2}{(t_1 - t_2)[t_2 + B'(t_1 - t_2)]} \tag{9}$$

De même, (6) se simplifie. Si on suppose $x_1 = a/b$ et $x_2 = a'/b'$ on a :

$$t_1 = (a-b)(a'+b') \qquad t_2 = (a+b)(a'-b') \qquad x_0 x_3 = \frac{(t_1 + t_2)^2}{(t_1 - t_2)^2} \tag{10}$$

On choisit la courbe de telle sorte qu'il existe un point Q de coordonnées a et b avec $3Q = 0$. Un tel point est facile à construire : il suffit que Q et $2Q$ ont la même coordonnée en x. En utilisant (5) on obtient :

$$B = \frac{-3a^4 - 6a^2 + 1}{4a^3} \qquad A = \frac{(a^2-1)^2}{4ab^2}. \tag{11}$$

Supposons n premier à 6, ce qui est une hypothèse raisonnable. Il faut que a soit premier à n, de sorte que A et B soient bien définis. On demande également que $a^2 - 1$ est premier à n, de sorte que A est non nul.

Soit P un point tel que $x = \pm 1$, et $P' = 2P$. On a alors $x' = 0$, donc $4P = 0$. Si un tel point existe sur la courbe, alors l'ordre du groupe est multiple de 12. Si on prend comme condition initiale un point dont l'ordre est premier à 6, son ordre sera $\leq L/12$, et on aura plus de chances de le calculer. La question est de savoir si le point P est sur la courbe. On suppose d'abord que p est premier à $A(B-2)(B+2)$. Si $A(B+2)$ est un carré modulo p^2, alors on a une solution $y = \sqrt{A(B+2)/A}$ pour $x = 1$. Dans le cas où $A(B-2)$ est un carré, on a une solution $y = \sqrt{A(B-2)}/A$ pour $x = -1$. Dans le cas où un et un seul de $B \pm 2$ est un carré, alors l'un au

moins de $A(B \pm 2)$ est un carré. Dans le cas contraire, $B^2 - 4$ est un carré, disons $B^2 - 4 = \Delta^2$. Alors $x = (-B \pm \Delta)/2$ et $x = 0$ sont trois racines de $y = 0$. On n'a donc pas de point d'ordre 4, mais trois points d'ordre 2.

Dans le cas où $B^2 - 4$ n'est pas un carré alors $y = 0$ a une racine simple. Dans le cas où c'est un carré, il y a trois racines. Le dénominateur de $B^2 - 4$ est $16a^6$, et le numérateur est $(a^2 - 1)^3(9a^2 - 1)$. On ajoute la condition que $9a^2 - 1$ est premier à n, c'est la condition qui dit que la courbe elliptique n'a pas de point double (rappel: on a déjà supposé que a et $a^2 - 1$ étaient premiers à n).

Il s'agit maintenant de trouver une condition initiale pour P_0. On choisit par exemple $x = 3a/4$. C'est une quantité non nulle. Pour qu'il existe y il faut que $Ax(x^2 + Bx + 1)$ soit un carré. En développant, on voit que $9 - 6a^2$ doit être un carré. On trouve que $a = 6r/(r^2 + 6)$ convient. Cette formule magique peut être obtenue de la façon suivante : on dit que $9 - 6a^2$ est le carré de $3(1 - 12/u)$. Tout nombre rationnel peut s'écrire sous cette forme. En développant, on voit que $u + 6 = (au/9)^2$. On prend donc $r = au/9$, et on en déduit a.

Algorithme 34. (Lenstra) *Toutes les variables et paramètres sont des entiers, sauf A, X et Z qui sont des tableaux d'entiers. On notera parfois A la valeur du tableau A en position i. Le but est de factoriser n, qui est un argument implicite de toutes les procédures.*

Procédure elliptic_add. *Paramètres a_x, a_z, b_x, b_z, c_x, c_z.* [Implémente l'équation 10]

1. *Poser $t_1 = (a_x - a_z)(b_x + b_z)$, $t_2 = (a_x + a_z)(b_x - b_z)$.*
2. *Réduire t_1, t_2 modulo n.*
3. *Calculer $d_x = c_z(t_1 + t_2)^2$, $d_z = c_x(t_1 - t_2)^2$.*
4. *Réduire d_x et d_z modulo n.*
5. *Rendre d_x et d_z.*

Procédure elliptic_double. *Arguments a_x, a_z, A.* [Implémente l'équation 9; la variable A contient la quantité notée B' dans l'équation.]

1. *Poser $t_1 = (a_x + a_z)^2$, $t_2 = (a_x - a_z)^2$.*
2. *Réduire t_1 et t_2 modulo n.*
3. *Poser $a_x = t_1 t_2$, $a_z = (t_1 - t_2) * (A * (t_1 - t_2) + t_2)$.*
4. *Réduire a_x et a_z modulo n.*
5. *Rendre a_x et a_z.*

Procédure elliptic_mul. *Paramètres N, M, a_x, a_z et A.*

1. *Poser $b_x = c_x = a_x$, $b_z = c_z = a_z$.*
2. *Poser $N = N - M$.*
3. *Appeler* `elliptic_double` *sur a_x, a_z et A.*
4. *Si $N > M$, mettre le résultat dans a_x et a_z, sinon dans b_x et b_z.*
5. *Si $N > M$ poser $N = N - M$, sinon $M = M - N$.*
6. *Tant que $M \neq 0$*
   - **6.1.** *Appeler* `elliptic_add` *sur a_x, a_z, b_x, b_z, c_x, c_z. Résultat d_x, d_z.*

6.2. *Si $N > M$ remplacer c_x, c_z par a_x, a_z, et a_x, a_z par d_x, d_z.*

6.3. *Sinon remplacer c_x, c_z par b_x, b_z, et b_x, b_z par d_x, d_z.*

6.4. *Si $N > M$, remplacer N par $N - M$, sinon M par $M - N$.*

7. *Rendre a_x et a_z.*

Procédure elliptic_power. *Arguments a_x, a_z, A, P, s, B_1.*

1. *Poser $p = P$.*

2. *Tant que $p \leq B_1$*

2.1. *Appeler* `elliptic_mul` *sur P, s, a_x, a_z et A.*

2.2. *Remplacer p par pP.*

3. *Rendre a_x, a_y.*

Macro lenstra_main. *Arguments s, p, C, B_1.* [Si un facteur de n a été trouvé, on sort de la procédure principale.]

1. *Soit S_p la partie entière de sp, $f = 1$.*

2. *Pour $0 \leq i < C$*

2.1. *Soit $a_x = X_i$, $a_z = Z_i$ et $A = A_i$.*

2.2. *Appeler* `elliptic_power` *sur a_x, a_z, A, p, S_p et B_1.*

2.3. *Remplacer f par le reste de la division par n de fa_z.*

3. *Remplacer f par le pgcd entre f et n.*

4. *Si $f \neq 1$, fin de la fonction, f est un facteur de n.*

5. *Sinon remplacer p par* `nextprime`(p).

Macro lenstra_par. *Argument i* [Calcule les paramètres de courbe en position i; avec de la chance, on factorise n.]

1. *Poser $a = 0$.*

2. *Poser $b = a$.*

3. *Choisir un nombre aléatoire r entre 0 et $n - 1$.*

4. *Soit $D = r^2 + 6$.*

5. *Soit G le pgcd de D et n.*

6. *Si $G \neq 1$ et $G \neq n$, fin de la procédure, G est un facteur de n.*

7. *Si $G = 1$, poser $a = 6r/D \bmod n$* [Bezout]

8. *Si $G = n$, ou si le a qu'on vient de calculer est égal à b, aller en 2.*

9. *Soit b le pgcd de n et $a(a^2 - 1)(9a^2 - 1)$.*

10. *Si $b = n$, calculer le pgcd entre n et a, $a-1$, $a+1$, $3a-1$, $3a+1$. Si on trouve un facteur non trivial, fin de la procédure, on a un facteur de n, sinon aller en* **2.**. *Si $b \neq 1$, fin de la procédure, b est un facteur de n.*

11. *Poser $b = ((-3a^4 - 6a^2 + 1)/(4a^3) + 2)/4 \bmod n$* [Bezout].

12. *Mettre b dans A_i, 1 dans Z_i et $3a/4 \bmod n$ dans X_i.*

Procédure Lenstra. *Argument* n [trouve un facteur non trivial de n, ou échoue et rend 0]

1. *Soit* $B_1 = 10^6$, $C = 30$. [d'autres valeurs sont possibles]
2. *Soient* A, X, Z *trois vecteurs de taille* C.
3. *Pour* i *entre 0 et* $C - 1$, *appeler* `lenstra_par`(i).
4. *Poser* $p = 2$, $s = 0.618...$ [inverse du nombre d'or].
5. *Tant que* $p < B_1$, *appeler* `lenstra_main` *sur* s, p, B_1 *et* C.
6. *Rendre 0.*

Bibliographie

[1] A. Galligo, J. Grimm, and L. Pottier, *The design of* sisyphe *: a system for doing symbolic and algebraic computations*, in LNCS 429 DISCO'90, A. Miola, ed., Springer-Verlag, Capri, Italy, Avril 1990, pp. 30–39.

[2] J. Grimm, *L'arithmétique générique de* sisyphe, Rapport Technique 136, INRIA, Dec. 1991.

[3] J. Hervé, F. Morain, D. Salesin, B. Serpette, J. Vuillemin, and P. Zimmermann, *BigNum: Un module portable et efficace pour une arithmétique à précision arbitraire*, Rapport de Recherche 1016, INRIA, 1989.

[4] A. Hurwitz, *Über die Entwiclung komplexer Grössen in Kettenbrüche*, Acta Mathematica, 11 (1888), pp. 187–200.

[5] ——, *Über eine besondere Art der Kettenbruch-entwicklung reeller Grösse*, Acta Mathematica, 12 (1889), pp. 367–405.

[6] ——, *Über die Kettenbrücher, deren Teilnenner arithmetische Reihen bilden*, Vierteljahrsschrift der Naturforchenden Gesellschaft in Zürich, Jahrg. XLI, Jubalband II (1896), pp. 34–64.

[7] INRIA, *Le_Lisp de l'INRIA Version 15.22, Le Manuel de Référence*, INRIA ed., Jan. 1989.

[8] D. Knuth, *The Art of Computer Programming*, Addison Wesley, 1981.

[9] P. L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation, 48 (1987), pp. 243–264.

[10] G. L. Steele Jr, *Common LISP: The Language*, Digital Press, Burlington, MA., 1984.

[11] J. Vuillemin, *Exact real computer arithmetic with continued fractions*, Rapport de Recherche 760, INRIA, Nov. 1987.

Index

Liste des algorithmes

Table des matières

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399